DOCUMENT RESUME

ED 330 326 IR 014 945

AUTHOR	Muir, Michael
TITLE	Fantastic Journey through Minds and Machines.
INSTITUTION	International Society for Technology in Education, Eugene, OR.
REPORT NO	ISBN-0-924667-74-5
PUB DATE	90
NOTE	135p.
AVAILABLE FROM	International Society for Technology in Education, 1787 Agate Street, Eugene, OR 97403-1923. Includes sample programs on data diskettes in three formats (LogoWriter for MS-DOS, LogoWriter for Apple II, and Terrapin Logo PLUS for Apple II).
PUB TYPE	Guides - Classroom Use - Instructional Materials (For Learner) (051) -- Guides - Non-Classroom Use (055) -- Computer Programs (101)
EDRS PRICE	MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS	*Artificial Intelligence; Computer Games; Computer Software; Educational Strategies; Elementary Secondary Education; Expert Systems; *Microcomputers; *Programing; Programing Languages; Robotics
IDENTIFIERS	*LOGO Programing Language

ABSTRACT

Intended for learners with a basic familiarity with the Logo programming language, this manual is designed to introduce them to artificial intelligence and enhance their programming capabilities. Nine chapters discuss the following features of Logo: (1) MAZE.MASTER, a look at robots and how sensors make machines aware of their environment; (2) SMARTTT, a discussion of games of strategy and decision making; (3) TALK.AND.DRAW, a description of languages for both humans and machines; (4) DOCTOR, a continuation of the discussion of languages that looks at conversationalists; (5) BLOCKS, a creation of a microworld in which a computer answers questions about work it has done; (6) INFER, an examination of logic, specifically, a hypothetical syllogism; (7) SUPER.INFER, a look at knowledge representations in intelligent programs; (8) SUPER.INFER2, a collection of procedures providing a natural language interface; and (9) VEXPERT, a discussion of expert systems. Each chapter contains four parts: background information, sample program, suggested extensions and improvements, and the complete program listing in LogoWriter and in Terrapin Logo PLUS. The LogoWriter MS/DOS data disk and LogoWriter Apple II data disk that accompany the manual are not included in this document. (21 references) (DB)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
Office of Educational Research and Improvement
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

☐ This document has been reproduced as received from the person or organization originating it.

☐ Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official OERI position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY

D. Moursund

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

BEST COPY AVAILABLE

About the Author

Mike Muir enjoys tinkering with both computers and education in central Maine. He has taught high school Mathematics and Computer Science, been Computer Coordinator for a small school district, and worked with teachers around the state as part of the Maine Computer Consortium. In 1988, his work with Logo and Artificial Intelligence earned him an Innovative Education Grant and a Gifted and Talented Grant, each from the State of Maine, and a University of Virginia/ LCSI Summer Logo Fellowship.

Currently, he enjoys his work as a LEGO TC Logo Teacher Trainer and as the Computer Resource Specialist at Skowhegan Area Middle School, one of two middle schools designated a Light House School by the Maine Department of Education for their work with restructuring. Mike is helping to lead intensive projects to integrate computers into the curriculum, to introduce multimedia tools, and to work with teachers on developing a vision for the future of education.

Occasionally, Mike spends time away from his computer with his wife Penny, two dogs (one of whom is named terrapin), two cats, and two horses. Mike and Penny are in the middle of a special project to create Artificial Intelligence from all natural ingredients, and are looking forward to their first child.

Project Editors: Sharon Yoder, Talbot Bielefeldt, Anita Best
Cover Design: Percy Franklin
Production: Tamara Kidd

For Ordering Information And A Complete Catalog:
International Society for Technology in Education
1787 Agate Street, Eugene, Oregon 97403-1923
Phone: 503/346-4414 Fax: 503/346-5890
CompuServe: 700014, 2114 Bitnet: ISTE@Oregon

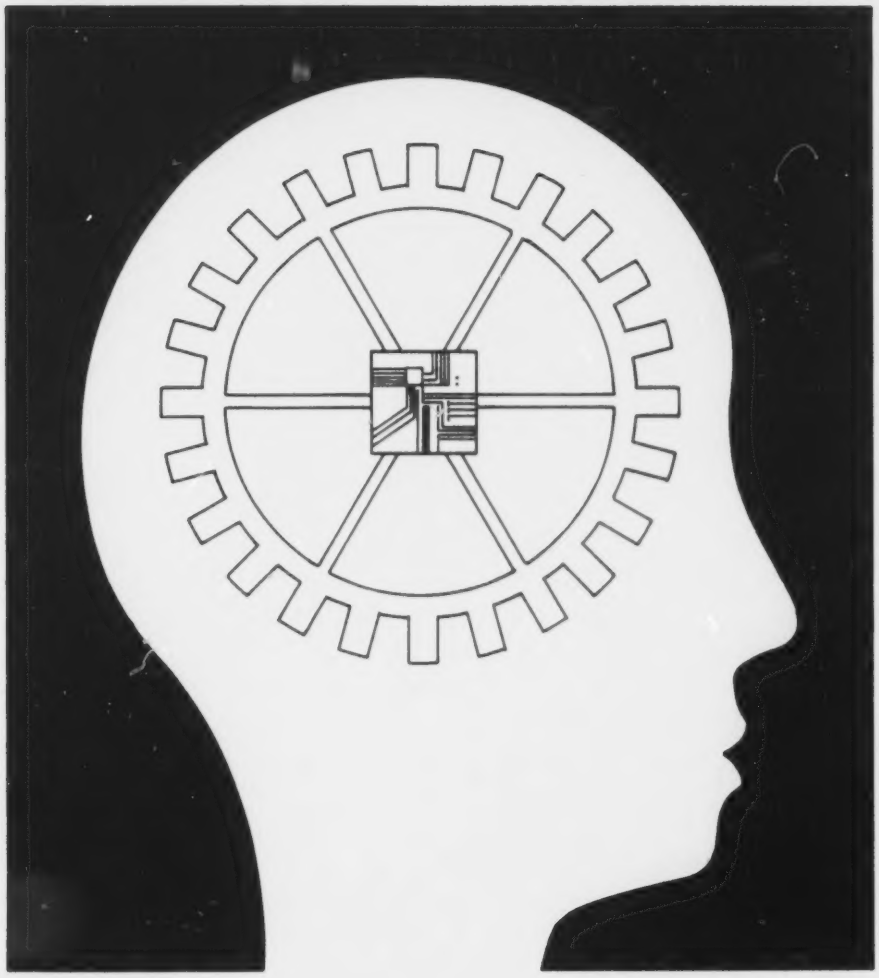
This publication has been produced using Aldus *PageMaker*®.

The text pages of this publication have been reproduced using recycled paper.

© 1990 ISTE
ISBN 0-924667-74-5

ISTE is a non-profit organization with its main offices housed at the University of Oregon.

Fantastic Journey Through Minds and Machines



by Michael Muir

© ISTE 1990

ISBN 0-924667-74-5

Acknowledgements

Any book is always the work of more people than just the author. Without the support and participation of many people, you would not be reading this book now. My name may be on the cover, but I am indebted to many others for bringing my vision into reality. Because this is my first book, I feel that I have more people to thank than if it were my third or fourth.

Many of my professors at Colby College sparked my passion for learning. Four had a special role in laying the groundwork for this book. I wish to thank John Hosack for nurturing my interest in computer science; the late Harold Jacobson for supporting my devotion to education and introducing me to educational technology; Jean Sanborn for teaching me how to write effectively; and Don Small for his continuing support, friendship, and mentorship.

Many of the chapters of this text grew out of courses in a Masters of Education program offered through Lesley College Outreach in Hampden (Bangor), Maine. Many of my colleagues in the program offered invaluable support and feedback. The instructors who read chapters also added to the text. Special appreciation goes to Craig Dickinson, who, in addition to encouragement, shared with me his own insights into writing and publishing.

Two chapters were developed through the University of Virginia Summer Logo Fellowship Program. LCSI and the Curry School of Education at the University of Virginia sponsored my studies there in July and August of 1988. I'd like to thank the "fellow Fellows," Mike Charles and Dee Miller, for their friendship, comradeship, and assistance during the Fellowship period. I would like to extend special thanks to Tom Lough, Glen Bull, Doris Flowers, Judi Harris, and all the others at UVa who made every resource available to the Fellows and who made the Fellows feel important for their three weeks in Virginia. Tom has continued to encourage and support my work, for which I am especially grateful.

Fantastic Journey would still be rough and untested if it weren't for the people involved in the Fantastic Journey Project at Winslow High School in Winslow, Maine. Superintendent Leon Duff, Principal Leroy Blood, and Mathematics Department Head Lester Lawrence gave me the freedom to try many new ideas, one of which was teaching gifted students about artificial intelligence. The Maine Department of Educational and Cultural Services supported the project in its first year through an Innovative Education Grant and a Gifted and Talented Grant. Eleven students from four area high schools participated in the course during its first year. The course used a draft of this book as the text. I appreciate the help of the students who offered their usual direct, no-punches-pulled critiques and suggestions. "My" kids, Fred Whitman, Adam Bulduc, Brian Shorey, and Chris Snipe lovingly offered me their brutally honest advice, which I found most valuable.

E. Paul Goldenberg read an early draft of *Fantastic Journey* and shared pages of invaluable suggestions. His insightful advice stays in the forefront of my mind, and I am sure it will shape my work in the future.

My editor, Sharon Yoder, put many hours into making this book a reality, and without her efforts and that of Anita

Best, Tamara Kidd, and the others at ISTE, this book would still be a collection of data disks on my shelf.

The people most influential in this undertaking were my family, especially my wife, Penny. She unselfishly gave me up on frequent weekends while I was away from home attending classes or conferences. She tolerated the times I was home when I locked myself in my office to work on the book. Meanwhile, Penny kept things running smoothly around the house when I let things slide, didn't yell too loudly when I printed late at night, and even managed to find the energy to encourage me and make me feel proud of what I had accomplished.

Contents

INTRODUCTION 1

1 MAZE.MASTER—Robots and Sensors 9
The Program 13
Possible Extensions & Expansions 15
LogoWriter Version Program Listing 16
Logo PLUS Version Program Listing 17

2 SMARTTT—Game Playing and Intelligence 19
The Program 23
Possible Extensions & Expansions 26
LogoWriter Version Program Listing 27
Logo PLUS Version Program Listing 29

3 TALK.AND.DRAW—Languages for Humans and Machines 33
The Program 35
Possible Extensions & Expansions 38
LogoWriter Version Program Listing 39
Logo PLUS Version Program Listing 41

4 DOCTOR—Conversationalists 43
The Program 46
Possible Extensions & Expansions 49
LogoWriter Version Program Listing 50
Logo PLUS Version Program Listing 51

5 BLOCKS—Machine Intelligence 53
The Program 55
Possible Extensions & Expansions 61
LogoWriter Version Program Listing 61
Logo PLUS Version Program Listing 65

6 INFER—Computers and Logical Inference 69
The Program 70
Possible Extensions & Expansions 75
LogoWriter Version Program Listing 75
Logo PLUS Version Program Listing 78

7 SUPER.INFER—
Knowledge Representaions in Intelligent Programs 81
The Program 83
Possible Extensions & Expansions 88
LogoWriter Version Program Listing 89
Logo PLUS Version Program Listing 91

8 SUPER.INFER2—A Natural Language Interface for Super.Infer 95
The Program 98
Possible Extensions & Expansions 100
LogoWriter Version Program Listing 101
Logo PLUS Version Program Listing 102

9 VEXPERT—and Expert Systems 105
The Program 112
Possible Extensions & Expansions 115
LogoWriter Version Program Listing 116
Logo PLUS Version Program Listing 118

Annotated Bibliography 121

Index 125

Introduction

The Journey Begins

When I was in junior high school, I was fascinated by magic. I read books on magic, watched magic shows on TV, and for Christmas, my parents bought me my first magic set. I mastered the tricks quickly, and they fired my desire to know more. Soon my folks and I had located several professional magic shops nearby, and my collection of magic tricks and books began to grow.

What had captivated my imagination was the illusion of the supernatural. I knew there was some trick to each illusion, but not knowing what that trick was heightened my interest. The mystery of how these tricks were performed and the search for the explanation of each illusion motivated me to learn all I could about magic. Although learning the tricks might have spoiled magic for me, it had quite the opposite effect. The fact that most of my friends didn't know how the tricks worked made me proud of what I knew, and whenever I saw a trick performed that I had never seen before, I really enjoyed trying to deduce how the trick worked.

When I was in college, I took a special interest in computers, and my fascination with magic was replaced with a preoccupation with artificial intelligence. Artificial Intelligence, often called AI, is the field of computer science concerned with creating programs that function in ways we would deem intelligent if a person where to behave in the same way. They are the programs science fiction is made of. The programs that use and appear to understand natural languages like English. Programs that control robots. Programs that make decisions, write their own programs, or play games. Some describe AI as trying to make machines do things that people currently do better.

To me, AI had the same supernatural feel that magic did. There is something magical about a program that can discuss your problems with you, can find its way out of a maze, or can deduce answers to questions for which it had never been told the answers. I knew there were programming tricks that explained how the programs worked and I wanted to know what those "tricks" were.

My search for the "tricks" to AI lead me to books about artificial intelligence. Most of the books I examined fell into one of three categories: general books about computers, artificial intelligence texts, and books about AI written for hobbyists. The information in general computer texts was limited, and AI texts were often very technical and provided more detail than I needed. There was a great deal of variation in the quality of books written for hobbyists. Most included short programs written in either BASIC or Pascal.

I share with you here some of the tricks I learned. The title, *Fantastic Journey Through Minds and Machines: Exploring Artificial Intelligence With Logo*, aside from being a mouthful, reflects some of the magic I discovered while on my quest to learn more about artificial intelligence. I was surprised to discover that many of the programming methods, although involved, were not complicated. I started thinking about what it meant to be intelligent and questioning some of the human behaviors I used to classify as intelligent. If machines could so easily mimic some of those behaviors, were the behav-

iors truly intelligent? As I learned how difficult (or impossible) it was to write programs to mimic other behaviors, I also began to marvel at the human machine with its grace, capacity, and complexity. I hope this book provides you with the same kind of magical journey I have enjoyed.

There are four parts to each chapter. Each chapter begins with background information on some AI topic, such as expert systems, robotics, game playing, human languages, and so forth. Next, a sample program is introduced. The key procedures are explained and discussed and sample output is included. The third section is a collection of suggested extensions and improvements the reader might make to the program. Last is the complete program listing.

Logo was selected as the language for the sample programs. Logo is a widely available, powerful language, and most people find getting started with Logo easy. Its primary data structure is the list, a dynamic structure that lends itself to working with groups of words and other symbols, as is necessary in much AI programming. Logo has many other features that simplify writing AI programs.

Logo is a cousin of LISP, the language most widely used for artificial intelligence in the United States. Logo is a procedural language, which means it is a natural environment for breaking up programs into small pieces. A modular program is often easier to test, debug, and expand than one that isn't. Logo is also flexible. Many languages require a strict structure and require the programmer to pay a great deal of attention to program syntax. Having a more flexible structure lets the programmer focus on the purpose and function of the program and not on the programming language.

Many versions of Logo have the added feature of being nearly "self-aware." Logo has primitives that allow a Logo program to create, examine, and manipulate another Logo program. A Logo program can even examine and modify itself. Also, variables can represent other variables, and a Logo program can change a value through several layers of variables. The variables and programs themselves become the data and can be modified as we modify our own thinking from time to time.

Each chapter has program listings in two Logo versions: LogoWriter from Logo Computer Systems, Inc. (LCSI) and Terrapin Logo PLUS (explanations in the chapters use the LogoWriter version). I have avoided using special features of any particular Logo version, so people using other versions of Logo should be able to adapt the programs without much difficulty (except graphics-only versions of Logo). I have also avoided using color commands for those people who do not have access to a color monitor. This book comes with the sample programs on data disks in three formats: LogoWriter for MS-DOS machines, LogoWriter for Apple II computers, and Terrapin Logo PLUS for Apple II computers.

Overview of Chapters

When I used to think of artificial intelligence, the first thing which came to mind were artificial people—robots. *Fantastic Journey* begins by exploring the lore and reality of robotics. You will learn of the myths and stories of artificial people. You will visit eighteenth-century mechanical people, and you will explore the industrial robots of today. You will learn how sensors make machines aware of their environment. The Logo turtle will be able to sense its way out of a maze with

the program in the first chapter, MAZE.MASTER.

The second chapter, SMARTTT, looks at games of strategy and decision making. Chess was an early focal point of AI research, even though chess programs are so commonplace today that we may not think of them as anything special. The research, however, has led to the development of programming techniques useful in applications requiring some sort of decision making. SMARTTT uses some of these techniques to play tic-tac-toe.

Science fiction is filled with computers and machines that can talk and be spoken to. Communicating with computers that understand human language is a dream of many people. Pieces of the dream are not far from reality. The third chapter looks at various kinds of computer languages, including natural language processors, those programs that allow a machine to understand English or other human languages. TALK.AND.DRAW is a program that will allow you to control the graphics turtle with English sentences.

The fourth chapter continues to look into the topic of natural language processors and specifically at conversationalists, programs designed to carry on conversations with people. Dr. Joseph Weizenbaum's program *Eliza* was a controversial example. It used programming tricks to give the appearance of understanding a conversation. DOCTOR is a version of Weizenbaum's *Eliza*.

Some AI experts feel that computer programs will only be intelligent when you can not only tell a computer about something but also ask it questions about that something. Some of those experts feel that one way to achieve this is by creating a microworld. Terry Winograd's SHRDLU controlled a robot arm that manipulated blocks of different shapes and colors. Not only could the program respond to English commands but also could answer questions about the work it had done. BLOCKS is a simplified version of Winograd's SHRDLU.

Another area of AI research is logical reasoning. Both the sixth and seventh chapters deal with this subject. The sixth chapter examines a form of logic called Hypothetical Syllogism. INFER is a program capable of performing this type of logic, which classifies objects. It can even deduce things it was never directly told! SUPER.INFER expands on INFER. SUPER.INFER can both classify objects and associate properties with those objects. The seventh chapter discusses the data structures needed to accomplish this.

The next chapter also expands on INFER and SUPER.INFER but does not focus on logical reasoning. SUPER. INFER.2 returns to the topic of natural language processing and understanding human languages. This chapter takes a look at natural language "front ends," programs that understand human language and are used to run other programs that don't. SUPER.INFER.2 is a collection of procedures to be added to SUPER.INFER so the program will understand English commands and inquiries.

The final chapter delves into the world of expert systems, one of the most commercial realms of artificial intelligence. Expert systems are used in a wide variety of fields, including aviation, medicine, law, education, and geology. Knowledge engineers create programs that act as experts. The engineers closely study human experts and how they make decisions in order to create programs to do the same. VEXPERT is a vacation expert system that will help you determine where to spend your next holiday.

A Few Words to the Learner

This book is designed to introduce you to the world of artificial intelligence and give you a chance to expand your programming capabilities. Although you can learn quite a bit from the background section of each chapter, some of the most interesting insights come from experimenting with the sample programs. The sample programs are intended to be a starting point, however. By using some of the suggestions in the book or using some of your own ideas, you can make the programs your own by extending and expanding them. Through this process, you will also learn a lot about programming and the techniques used in artificial intelligence.

This book assumes that you are already familiar with your version of Logo. You should know how to create and modify a procedure as well as save and load files or pages. You should also be familiar with turtle graphics commands and word and list processing commands. Having a basic understanding of recursion is also very helpful. The more experienced you are with programming, the more comfortable you will be with expanding the programs. The hints below should also assist you as you "hack around."

The section on how the program works can be hard to read before you have entered the program and used it for a while. Read that section after you have used the program. Although using the programs on the data disk is tempting, you will gain much more understanding of how each program works by typing it in yourself. This is important if you are going to try programming some extensions or expansions.

If you get stuck while extending a program, look to the sample programs for programming ideas. Just as you can get building ideas by looking at houses and writing ideas by reading other people's work, you can get ideas for programming by examining other people's programs.

Use all the resources you can to help you write the program you want. Talk to a programming expert you know, your computer science teacher, or the student working at the next computer. Keep books about Logo handy (see the References section at the end of the book). Just because you want it to be your program doesn't mean you have to work alone.

Share what you create with others. There are two good reasons for this. Other people will want to see what you have done to get ideas for their own work. Also, after putting in so much hard work, you deserve some credit!

Some Thoughts on Teaching With This Book

This is a book for people like me who are captivated by the magic of AI and want to know more. It is also for people who want to use the magic of AI to motivate students. *Fantastic Journey* was designed to be used a variety of ways. It is first a text about artificial intelligence. With it, learners can read about several fields of AI research and enter and modify programs that are examples of programming in these fields. I found when teaching this course that it was very important in classroom discussions to keep coming back to the key procedures to highlight the programming techniques utilized. This way students quickly grasp AI programming methods.

Fantastic Journey is not just a text about artificial intelligence but also about advanced uses of Logo. Many teachers and students never get beyond turtle graphics or simple list processing. Those who are interested can now go beyond

and work with some examples of powerful uses of Logo. Because of its association with young children, Logo is often thought of as a simple language. Many people do find it easier to learn than other programming languages, but that doesn't mean that the language is not powerful. More and more books that explore Logo's potential are now being published.

Fantastic Journey isn't just a text in the sense of being a collection of knowledge. It is a project book. The programs in each chapter can be used as functioning examples, but they can also be used as jumping off places for larger projects. Advanced programming and artificial intelligence are not simple subjects, and starting a program from scratch without a model can be frustrating and at times seem impossible. The sample programs are intentionally skeletal to give the programmer a starting place for building a more full-featured program. The suggested improvements and extensions are an important feature and can help provide direction to programmers looking for ideas.

Educators can use the programs to various extents within their computer classes. The programs can simply be used as examples of AI in demonstrations when talking about different fields of computer science. An individual chapter can be examined when talking about AI. Students can enter the program and try it for themselves and maybe expand and refine the program. Some teachers might just focus on a few of the chapters. One educator might cover just one chapter, have the students examine the sample program, and then have them build their own, greatly expanded version. Other teachers might use the text for a course and have students go through all the chapters.

If you plan to use only a few chapters, TALK.AND.DRAW, BLOCKS, and VEXPERT are my favorites. They are interesting examples of AI, and the programming used is not too complex. Beware of SUPER.INFER and SUPER.INFER.2 if your students do not have a good computer science background. These two chapters are more technical. MAZE.MASTER, the chapter on robotics and sensors, makes a good introduction to units on robotics, especially if you plan on using products like Fischer Technik or *LEGO TC logo*.

The programs in the text are aimed at the intermediate programmer. Students should be familiar with your version of Logo. They should know how to create and modify a procedure as well as how to save and load files or pages. Students should also feel comfortable with turtle graphics commands and word and list processing commands. Having a basic understanding of recursion is also very helpful.

If you do plan on using *Fantastic Journey* for a programming course, here are some tips and suggestions for lesson design, your role, and student assessment. They reflect how I have taught with the text and are offered only as hints. You should use the text as it best fits your teaching style.

When teaching, I use *Fantastic Journey* as a project book. A project-oriented classroom can resemble a circus! Students work together programming, designing, testing, and redesigning their projects. Groups interact with each other and ideas spread throughout the classroom. A project-oriented approach offers context and motivation to using problem solving strategies and programming techniques. It also helps students to understand the process of designing and implementing a program.

To prepare students for working on

their projects, have them read the first section of the chapter they are working on. This will provide them with background on the field of AI they are studying. Supplement this with films, other texts, or field trips to local universities, museums, or AI firms.

Next, have students type and debug the sample program. (Depending on the students' typing speed and the length of the program, each program takes a class session or two to enter.) This helps them understand how the sample program works. Students can use the data disk instead of entering the program themselves, although I found that students who entered the program understood it more easily. You could demonstrate the program as students are working so they can see how to use the program and how it is supposed to function once it is debugged. (Each program should be bug free as listed, but typing errors and using different versions of Logo can cause some difficulties.) Have students read the section in the book describing how the program works and then lead a discussion to check their understanding. Students still having difficulties may find it useful to create a procedure tree for the program. It will be very hard for students to go on to their own projects if they don't understand the sample program.

Spend a little time focusing on the programming techniques, data structures, and control structures used. This will facilitate students using the techniques and structures in their own programs.

I allocated most of the class time to be spent having students modify and expand the programs. The section on possible extensions and expansions in each chapter should be helpful. Students who wish to rewrite the program completely should be encouraged to do so. Group work should also be encouraged, since most larger "real-life" programs are written by teams of programmers.

After students have improved their programs, team members should share their work with the class. They should discuss what they set out to do, what they accomplished, how they did it, and what problems they ran into. This reflective time allows everyone to think about what he or she has learned while working on the project. By thinking about the projects and discussing experiences, students can relate experiences in one instance with experiences in another.

In the project-oriented classroom, it is important to give students the opportunity to explore and discover on their own. It is also important to provide some structure to the discovery process. You can support students in several ways. As "Knowledge Giver," you introduce new ideas and motivate students to explore new areas. As "Discussion Leader," you encourage students to share their discoveries and views. Through the discussions, you can connect experiences and learning to other projects, curriculum areas, and real world concepts. As "Consultant," you help students debug programs, find resources, and connect with other students. As a result, students develop better problem solving and independent learning skills.

Don't feel that you must have all the answers to student questions. Having students get answers from other students is good for both sets of students. You will also learn from your students. Raising the question "What are some ways we could try to find the answer?" can help students become better problem solvers. Be more concerned with supporting each student in his or her quest for an answer than with providing that answer.

There is a lot of discussion around the issue of evaluating student work in a project-oriented classroom. It is tempting, although a bit deceiving, to evaluate a student's learning based on the finished product. More accurate assessment comes when a student's process, instead of the result, is the focus. It is also important to evaluate each student individually rather than to compare students with others in the class.

One tool for evaluation is having students keep a journal. Students can use their journals as a place to log projects they attempted, problems they experienced, questions they had, and solutions they found. These thoughts, together with a program listing and sample run of the program, can provide you with insight into each student's learning and problem solving process.

Asking students a few questions can also help you with evaluating student work and understanding of the process he or she followed. If the answers to some of these questions are going to be used for evaluation, you might want to keep a log of your conversations with students. Here are some questions to try:

- What is the problem you are trying to solve?
- Could you tell me how you use your program?
- What other ideas did you have at the beginning?
- Can you tell me what special procedures you have used?
- How does this procedure work within your program?
- What questions have you thought about as you programmed this project?
- What aspects of this project are you most curious about?
- How well do you think it works?
- What improvements would you like to make?

Here are some other criteria to consider when evaluating student progress:

- Does the student work well with other students?
- Does the student share ideas and materials with others?
- Does the student have the discipline to work individually?
- Does the student become frustrated when things don't work right away?
- Does the student come up with creative ideas for projects?
- Does the student use "errors" as a source of new ideas?
- Does the student follow through and finish projects?
- Does the student explore ideas in depth?
- Does the student "experiment" with Logo?
- Does the student reflect upon the meaning of things that happen?
- Does the student understand "big ideas," such as natural language and recursion?

However you choose to use *Fantastic Journey*, I hope you find it interesting and fun. I encourage you to play with each program, and don't be afraid to try new ideas you might have about modifying them. Happy Hacking!

1 MAZE.MASTER— ROBOTS AND SENSORS

The possibility of creating a machine that looks and behaves like a person has been a dream that has helped to drive AI research. Even though artificial intelligence researchers are beginning to assemble the tools to create artificial people, the vision is not a new one. History, myth, and fiction are alive with examples.

Several ancient civilizations, including the Egyptians and the Chinese, created animated statues. These could make lifelike movements and were often hydraulically powered. Hero of Alexandria created mechanical birds that could fly and sing (Albus, 1981). During the eighteenth and nineteenth centuries, automated humanlike machines were very popular. One popular automaton was taken on tour around Europe in the 1800s. It could play chess against a human opponent and appeared to be a fairly competent player. Edgar Allen Poe finally uncovered the fraud, exposing that a small man was really making all the chess moves from inside the machine.

Still on exhibit and in working condition at the Musee d'Art et d'Histoire in Neuchatel, Switzerland, are three special mechanical people. The Scribe sits at a small table and writes on paper. The letters are well-formed and at the end of each line, the Scribe lifts his pen and dips it in the ink well. Even his eyes follow his writing. The Draughtsman looks like the Scribe but instead of writing can create any of four drawings. The Musician plays a miniature organ. Her fingers move over the keys, and her body, head, and eyes move much as a real organ player's would. Her chest even rises and falls regularly to simulate breathing. Each of these automatons is controlled by cams and gears and were created during the late 1700s by Pierre and Henri-Louis Jacquet-Droz.

One of the better known legends about artificial people concerns the Rabbi Loew of Prague and his Golem. In sixteenth-century Europe there was a strong anti-Semitic movement. Rabbi Loew was able to diminish the anti-Semitic feelings in his region with his work and teachings, until a Catholic priest named Thaddeus rekindled the hatred. One night, Loew had a dream telling him to create a golem of clay. Golems are artificial men created from the earth. The sacred name of God written on parchment and placed in the golem's mouth was supposedly one way to bring a golem to life. Loew built an eight-foot tall artificial person he named Joseph Golem. According to the legend, Joseph worked in the temple, patrolled the streets, and in general protected the Jews. Later, he was deactivated. One story says that the parchment was removed from Joseph's mouth because the Jews were now safe and he was no longer needed. Another story reports that Joseph went on a rampage and attacked innocent people, including the Rabbi Loew.

Until recently, much of the fiction about artificial people illustrated a fear of lack of control over artificial life. Mary Shelley's *Frankenstein* is a classic example. The playwright Karel Capek, who coined the word robot, also wrote about androids gone amok. In 1921, Capek, also from Czechoslovakia, like Rabbi Loew, wrote "Rossum's Universal Robots." "Robot" is from the Czech word for worker. As a scientist, Rossum creates robots to help people. They take on a dark side, however, when they are used to kill people in

war. Later, an irresponsible scientist gives the robots emotions and feelings. Finally, tired of being oppressed and controlled by people, the robots rebel and kill all humans.

In the 1940s, author Isaac Asimov started to change fiction's image of robots. He created worlds in which robots had to adhere to The Three Laws of Robots:

1) A robot may not injure a human being or through inaction allow a human being to come to harm.

2) A robot must obey the orders given it by human beings except where such orders would conflict with the first law.

3) A robot must protect its own existence as long as such protection does not conflict with the first or second law.

Many authors have adopted Asimov's Robot Laws but because the military is funding most current robotic and AI research, it is doubtful that real robots will follow the laws.

Adherence to Asimov's laws is not something that we must currently be concerned with. Many of today's robots are clunky one-armed beasts that are replacing the human work-force in tedious and hazardous jobs. Even if they work no faster than humans, industrial robots save companies money by not requiring coffee and lunch breaks or needing days off for illness or vacation. Robots have replaced people in dangerous environments, such as spray-painting and welding auto bodies or working with chemicals and radioactive elements. They have also been used effectively in repetitious and tedious jobs people find extremely dull, such as machining fine parts or fitting precision components.

David Peat (1988) gives an example of the effectiveness of industrial robots, even in small companies.

> TI Creda Ltd. in the United Kingdom manufactures ovens which require a vitreous enamel coating. The paint itself is too abrasive and unpleasant to make spraying an attractive job for humans so the ovens used to be hand dipped into a bath. The work was heavy and the coating itself tended to be uneven and to clog previously drilled holes which then had to be cleared manually. The company's solution therefore was to purchase a robot and train it to spray enamel. The process worked well and soon there were two robots, working side by side during an 18-hour workday. Within 2 years the robots had recovered their cost of purchasing and installation, were saving the company 40 percent in enamel, and were turning out an improved product.

Industrial robots frequently take the form of an articulated arm. The arm is jointed in one or more places and has a wrist that can be moved up and down as well as side to side. Although some robot arms have a hand or claw, others have multiple attachments that can be joined either automatically or manually to the wrist. Early industrial robot arms were pneumatically controlled, driven either by human remote control or programmed by plugging hoses into different air nozzles. Today's robots are much more sophisticated. Their mechanical joints are remote-controlled by microprocessors. With the addition of visual or touch sensors and decision-making software, a robot arm can function in a wide variety of situations.

Complex programs control the joints for coordinated movement on the assembly line. The software directs which tools to attach to the robot wrist, which manufactured parts to select, and which method should be used to assemble them. This type of coordination takes a great deal of computing power. One arm can be readily controlled by today's technology, but the coordinated control of two arms working together is still in the future.

Sensors can increase the versatility of a robot. Touch sensors can be used for detecting pressure. With their use, mobile robots can tell when they have bumped into a person or object. Robot arms used for assembling parts also can use pressure sensors. When precision parts are put together, alignments must be exact. If either of the two parts is slightly out of line, one piece may not fit into the other. A robot without sensors may try to force the parts even when they don't fit–a potential disaster. Sensors allow the robot to detect when one piece meets with resistance. Under software control, the robot responds by either moving the piece and trying to find the proper position or by stopping and sending a message to a human worker who can correct the problem.

Robot vision systems aid with the identification of parts. A "blind" robot must rely on parts being in exactly the same position each time the part is needed. Even having a part turned in the wrong direction or flipped on its side can result in a production breakdown and subsequent delay. A "seeing" robot can identify a part even if it is not in an exact location or orientation. Many of the current vision systems are of two types. One method relies on contrasting colors to define the edges of an object. Another method uses laser rangefinding. Each of the two types of systems sends its data to the microprocessor, which then tries to match the image to one of many 3-D mathematical "pictures" it has in memory. This takes a great deal of processing time but is simplified by the fact that for any one application, the computer needs to be familiar with only a few parts. When a robot is changed to a new job and reprogrammed, new mathematical formulas for pictures of the new parts can be loaded into the microprocessor's memory.

The commercial robots of today are limited to visual and touch sensors. Future machines may incorporate other types of sensing equipment. Radar, infrared, and ultrasonic systems may be used, as well as equipment that can detect chemical traces or electronic and magnetic fields. Although these types of sensors would help reach the goal of creating human-like robots, the limitations of mechanical design, computing power, and software necessary for intelligent, independently controlled limbs and body leave the development of such androids far in the future.

Several institutions, including Stanford Research Institute (SRI) International and Carnegie-Mellon University, are studying the creation of general robots. General robots are those that can move and complete tasks under their own direction. SHAKEY, built by SRI, has a variety of features, including vision, a rangefinder, bump sensor, and a wheeled drive system. Carnegie-Mellon's CART also uses vision and a wheeled drive system. CART makes up for the lack of stereo vision by moving its one eye from side to side on a track. General robots move by "looking" at their environment, constructing a map of their surroundings, moving, and then repeating the process.

Some research projects involve the development of robots that move on jointed

legs. As with robot arms, there are difficulties with coordinating the motion of two, four, or six legs. Legged robots present the additional challenge of coping with balance, especially on uneven surfaces. This aspect of the general robot research offers possibilities to handicapped populations. Some research facilities are studying the design and creation of highly sophisticated artificial limbs. The small electrical discharges from muscles can be picked up by electrodes and then used to drive mechanical joints. Other research has produced a network of electrodes that can be placed over healthy muscle that lacks the nerves necessary to be controlled by the brain. Computers can then be used to help bridge the neural gap by receiving the electrical signal before it reaches the damaged nerves and then transmitting the signal directly to the muscle. This technology is already being used to help people paralyzed by certain kinds of nerve damage to walk again.

Simple robots are becoming available to the consumer. Many toy manufacturers market kits of construction blocks or building components that can be used to build and operate robots. Fischer Technik sells an interface box and computer software with some of its kits. With the interface, machines built with Fischer Technik components can be attached to the computer and controlled with programs written by the user. The package is called the Robotic Laboratory and uses a language called PaRCL, which is a derivative of Standard Forth. The language has the advantage of being similar to Logo as well as Forth, a popular language for assembly-line robot control.

LEGO also has similar equipment. Among the 1,200 different types of LEGO blocks are motors, gears, lights, light sensors, and touch sensors. There are at least two special interface boxes that can be used to control LEGO creations with programs. These sets should be of interest to Logo programmers since each uses a version of Logo. One system is marketed by Terrapin and allows LEGO creations to be controlled with BASIC or Terrapin Logo. Another system marketed by LEGO uses a specially adapted version of LogoWriter. The system is called *LEGO TC logo*.

Included in the *LEGO TC logo* Starter Packs are plans for creating a variety of machines and ideas on how to use these machines. The directions on one set of activity cards can be used to create a conveyor belt and a robot arm that can be used to move blocks from the conveyor. Another activity card has the design for a robot turtle. Special *LEGO TC logo* procedures are included so that the LEGO turtle can be programmed to go left, right, forward, or back. A pen can be inserted into the turtle, and it can then draw on paper just as the Logo turtle draws on the graphics screen.

Perhaps more interesting than a turtle that moves is a turtle that can sense its environment and then act based on what it observes. When attached to the turtle, or any moving robot you create, LEGO's touch and light sensors allow you to send feedback to the computer. The light sensor detects a change from light to dark or from dark to light. If attached to the front of the turtle, the light sensor can be used to follow a black line on a white floor. A booklet that comes with the *LEGO Starter Kit* even suggests some procedures that will do this. If touch sensors are attached to the turtle, it can tell when it bumps into something, and the program can tell the turtle how to react.

One application of the turtle and its sensors might be to have the turtle find its way out of a maze. If touch sensors

were used, a walled maze would be necessary. The touch sensor could tell when the turtle bumped into a wall, and the turtle could then turn and look for a path in a new direction. Light sensors would allow mazes to be mapped out in black ink on white paper or with electrician's tape on a white floor.

The Program

MAZE.MASTER is a program which, after you create a maze, will direct the regular Logo graphics turtle to find its way out of the maze. Once the turtle has learned how to find its way through that maze, the program remembers the route and can then quickly retrace the path. Other authors have published programs similar to this one (e.g., Weston, 1985; Terrapin, 1988).

Our mazes will be made up of colored "alleys." The turtle already has commands built in for locomotion, so it has the means to move about the maze. What will keep the turtle from just wandering all over the screen? Many versions of Logo have a primitive called DOT? or DOTP that tells whether or not the turtle is positioned over a colored spot. This is analogous to a robot having an optosensor to see if it is positioned over a colored area. LogoWriter does not have such a primitive, but it is easy to create. DOT? outputs TRUE when the color under the turtle is not the same as the background color (when it is on the maze) and outputs FALSE when the background color is the same as that under the turtle (when it has wandered off of the maze).

```
TO DOT?
OUTPUT NOT (COLORUNDER = BG)
END
```

The turtle is controlled by a procedure called MOVE (see the listing below) that step-by-step sends the turtle down an alley until it finds the other end. When the turtle travels past the end, DOT? will output FALSE, and MOVE outputs the length of the alley back to an earlier procedure so it can be saved in :ROUTE. MOVE also calls FIND.DIRECTION to discover which way the turtle ought to turn for the next alley.

FIND.DIRECTION looks to the left and then the right. It locates the next branch of the maze by turning, then moving ahead and using DOT? to see if it is once again over a colored area. FIND.DIRECTION then outputs the directions for the turn so that it can be recorded in :ROUTE.

```
TO MOVE :DIST
IF NOT DOT? [BACK 4 OUTPUT ( SENTENCE
  [FORWARD] ( :DIST - 4 )
  FIND.DIRECTION [WAIT 20] )]
FORWARD 1
OUTPUT MOVE :DIST + 1
END

TO FIND.DIRECTION
LEFT 90
FORWARD 7
IF DOT? [BACK 7 OUTPUT [LEFT 90]]
BACK 7
RIGHT 180
FORWARD 7
IF DOT? [BACK 7 OUTPUT [RIGHT 90]]
BACK 7
RIGHT 90
OUTPUT [RIGHT 180]
END
```

MAZE.MASTER is the procedure that makes it all work. MAZE.MASTER takes as input the name of the procedure to draw the maze you wish to solve. Before MAZE.MASTER tries to solve it, the procedure calls SETUP. SETUP draws the requested maze by using the RUN primitive. RUN will execute a list of Logo commands whether they are primitives or

user-defined procedures. The value of :MAZE is the name of the procedure of the maze you wish to have drawn. SENTENCE makes it into a list. The parentheses allow SENTENCE to take less than the usual two inputs.

SETUP also initializes the variable :ROUTE so that it can be used to remember how to travel the maze. After solving the maze, you can have the turtle run the maze again by typing RUN :ROUTE. All of the Logo commands for navigating the maze will have been stored in :ROUTE.

```
TO MAZE.MASTER :MAZE
SETUP :MAZE
SOLVE.MAZE
END

TO SETUP :MAZE
HOME
RUN ( SENTENCE :MAZE )
MAKE "ROUTE [HOME WAIT 30]
END
```

SOLVE.MAZE starts by seeing where the turtle is. The turtle has solved the maze and found its way out if its X-coordinate is farther than 130 units from HOME or the Y-coordinate is farther than 85 units. (For the Logo PLUS version, the numbers are 100 units and 100 units.) Keep this in mind when you create mazes. Next, SOLVE.MAZE adds to :ROUTE by calling the procedure MOVE. Recall that MOVE uses DOT? as its visual sensor to find the end of an alley, then calls FIND.DIRECTION to find which way to go next.

```
TO SOLVE.MAZE
IF ( FIRST POS ) > 130 [STOP]
IF ( FIRST POS ) < -130 [STOP]
IF ( LAST POS ) > 85 [STOP]
IF ( LAST POS ) < -85 [STOP]
MAKE "ROUTE SENTENCE :ROUTE MOVE 0
SOLVE.MAZE
END
```

In order for the turtle to search its way through a maze, you must, of course, first create a maze for the turtle to search. Since a maze is nothing more than a collection of alleys of varying lengths that turn off from one another at right angles, all we need is a procedure to draw a rectangular alley. ALLEY takes as input the length you wish the alley to be. It then draws a rectangle of that length and fills it.

```
TO ALLEY :DIST
LEFT 90
FORWARD 4
RIGHT 90
FORWARD :DIST
RIGHT 90
FORWARD 8
RIGHT 90
FORWARD :DIST
RIGHT 90
FORWARD 4
RIGHT 90
PU
FORWARD ( :DIST - 4 )
PD
FILL
END
```

Short mazes can now be created just by using ALLEY together with LEFT and RIGHT. For example, ALLEY 30 RIGHT 90 ALLEY 40 LEFT 90 ALLEY 30 creates an arch-shaped maze. Mazes can be saved into procedures so that you can create libraries of mazes for the turtle to solve. MAZE1 is a good maze to start with. As you can see, the maze drawing procedures get a bit long.

```
TO MAZE1
START.MAZE
ALLEY 30
RIGHT 90
ALLEY 40
LEFT 90
ALLEY 20
LEFT 90
ALLEY 50
RIGHT 90
```

```
ALLEY  45
LEFT  90
ALLEY  20
LEFT  90
ALLEY  120
LEFT  90
ALLEY  90
LEFT  90
ALLEY  90
RIGHT  90
ALLEY  30
RIGHT  90
ALLEY  40
LEFT  90
ALLEY  50
FINISH.MAZE
END

TO  START.MAZE
CC
CG
SETC  3
PD
HT
END

TO  FINISH.MAZE
PU
HOME
SETC  1
ST
END
```

I use BLANK.MAZE as a template for creating new maze procedures. It already has all of the commands in START.MAZE for cleaning up the screen. All you have to do once you have BLANK.MAZE in the editor is change the name of the procedure and add the ALLEY and turn commands. It simplifies things greatly if you always start a maze in the home position. The commands in BLANK.MAZE take care of this for you. Now, test your new maze to make sure the alleys don't cross. Crossed alleys won't fill properly and MAZE.MASTER will not detect turnoffs before an alley ends. Alleys shorter than 20 units can also cause problems and should be avoided.

```
TO  BLANK.MAZE
START.MAZE
  insert  maze  commands  here
FINISH.MAZE
END
```

Note: Logo PLUS programmers may notice that there are a large number changes in the numeric values in procedures from their LogoWriter counterparts. These changes are needed because of the differences in the graphics screens of the two versions.

Possible Extensions and Expansions

- You will probably grow tired of MAZE1 fairly quickly. Create more mazes. You are limited only by the rule that states the maximum X- and Y-coordinates for a maze.
- Perhaps it would be more interesting to have other ways to end a maze. One possibility would be to have the turtle run from the outer part of the screen and end when the turtle reaches the center.
- MAZE.MASTER will only run simple mazes. That is, the turtle cannot recognize branches in the middle of an alley. The turtle is capable of finding dead ends by going to the end and turning around and coming back. See if you can find ways of teaching the turtle to follow more complex branching mazes.
- Use *LEGO TC logo*, Fischer Technik or another building blocks kit to create a robot to use light sensors, touch sensors, or some other type of sensor to run mazes.

MAZE.MASTER
THE PROGRAM
LogoWriter Version

```
TO ALLEY :DIST
LEFT 90
FORWARD 4
RIGHT 90
FORWARD :DIST
RIGHT 90
FORWARD 8
RIGHT 90
FORWARD :DIST
RIGHT 90
FORWARD 4
RIGHT 90
PU
FORWARD ( :DIST - 4 )
PD
FILL
END

TO MAZE1
START.MAZE
ALLEY 30
RIGHT 90
ALLEY 40
LEFT 90
ALLEY 20
LEFT 90
ALLEY 50
RIGHT 90
ALLEY 45
LEFT 90
ALLEY 20
LEFT 90
ALLEY 120
LEFT 90
ALLEY 90
LEFT 90
ALLEY 90
RIGHT 90
ALLEY 30
RIGHT 90
ALLEY 40
LEFT 90
ALLEY 50
FINISH.MAZE
END

TO START.MAZE
CC
CG
SETC 3
PD
HT
END

TO FINISH.MAZE
PU
HOME
SETC 1
ST
END

TO MAZE.MASTER :MAZE
SETUP :MAZE
SOLVE.MAZE
END

TO SETUP :MAZE
HOME
RUN ( SENTENCE :MAZE )
MAKE "ROUTE [HOME WAIT 30]
END

TO SOLVE.MAZE
IF ( FIRST POS ) > 130 [STOP]
IF ( FIRST POS ) < -130 [STOP]
IF ( LAST POS ) > 85 [STOP]
IF ( LAST POS ) < -85 [STOP]
MAKE "ROUTE SENTENCE :ROUTE MOVE 0
SOLVE.MAZE
END

TO MOVE :DIST
IF NOT DOT? [BACK 4 OUTPUT ( SENTENCE
  [FORWARD] ( :DIST - 4 )
  FIND.DIRECTION [WAIT 20] )]
FORWARD 1
OUTPUT MOVE :DIST + 1
END

TO DOT?
OUTPUT NOT (COLORUNDER = BG)
END
```

```
TO  FIND.DIRECTION
LEFT  90
FORWARD  7
IF  DOT?  [BACK  7  OUTPUT  [LEFT  90]]
BACK  7
RIGHT  180
FORWARD  7
IF  DOT?  [BACK  7  OUTPUT  [RIGHT  90]]
BACK  7
RIGHT  90
OUTPUT  [RIGHT  180]
END
```

MAZE.MASTER
THE PROGRAM
Terrapin Logo PLUS Version

```
TO  ALLEY  :DIST
   LEFT  90
   FORWARD  4
   RIGHT  90
   FORWARD  :DIST
   RIGHT  90
   FORWARD  8
   RIGHT  90
   FORWARD  :DIST
   RIGHT  90
   FORWARD  4
   RIGHT  90
   PU
   FORWARD  ( :DIST - 4 )
   PENDOWN
   FILL
END

TO  START.MAZE
   CLEARTEXT
   DRAW
   PENDOWN
   HIDETURTLE
END

TO  FINISH.MAZE
   PENUP
   HOME
   SHOWTURTLE
END
```

```
TO MAZE1
  START.MAZE
  ALLEY 30
  RIGHT 90
  ALLEY 40
  LEFT 90
  ALLEY 20
  LEFT 90
  ALLEY 50
  RIGHT 90
  ALLEY 45
  LEFT 90
  ALLEY 20
  LEFT 90
  ALLEY 120
  LEFT 90
  ALLEY 90
  LEFT 90
  ALLEY 90
  RIGHT 90
  ALLEY 30
  RIGHT 90
  ALLEY 40
  LEFT 90
  ALLEY 50
  FINISH.MAZE
END

TO MAZE.MASTER :MAZE
  SETUP :MAZE
  SOLVE.MAZE
END

TO SETUP :MAZE
  HOME
  RUN ( SENTENCE :MAZE )
  MAKE "ROUTE [HOME WAIT 60]
END
```

```
TO SOLVE.MAZE
  IF XCOR > 100 THEN STOP
  IF XCOR < - 100 THEN STOP
  IF YCOR > 100 THEN STOP
  IF YCOR < - 100 THEN STOP
  MAKE "ROUTE SENTENCE :ROUTE MOVE 0
  SOLVE.MAZE
END

TO MOVE :DIST
  IF NOT DOT? THEN BACK 5 OUTPUT
  (SENTENCE [FORWARD] ( :DIST - 4 )
  FIND.DIRECTION [WAIT 20] )
  FORWARD 2
  OUTPUT MOVE :DIST + 2
END

TO FIND.DIRECTION
  LEFT 90
  FORWARD 8
  IF DOT? THEN BACK 8 OUTPUT [LEFT 90]
  BACK 8
  RIGHT 180
  FORWARD 7
  IF DOT? THEN BACK 7 OUTPUT [RIGHT 90]
  BACK 7
  RIGHT 90
  OUTPUT [RIGHT 180]
END
```

2 SMARTTT— GAME PLAYING AND INTELLIGENCE

If a machine can play a sophisticated game like chess, is it an intelligent machine? Many of the AI pioneers thought so. Today, it is hard to think of computer chess games as intelligent since there are so many available. In the 1940s and 1950s, however, the question "Is the machine intelligent?" was like asking if a machine can "hear" and "speak" English and discuss literature with a human. Chess playing was a task that, at the time, computers were unable to perform. More importantly, chess was a skill that was thought to require human qualities like intellect, cunning, creativity, and intuition. Leaders in AI research felt that if they could program a machine with the qualities to play chess or other games of strategy, then the machines would be intelligent. Creating game-playing programs was one of the first ways that people attempted to build artificially intelligent machines.

In the mid-1950s, Claude Shannon of Bell Labs, Marvin Minsky of MIT, and John McCárthy of Dartmouth worked together to arrange a conference on artificial intelligence. The conference was finally held at Dartmouth in 1956. The Dartmouth Conference was the first computer conference at which the issues and possibilities of artificial intelligence was discussed. The term artificial intelligence was coined at this conference, and the conference signaled the birth of AI as a field of computer science. At the time, several people were already working on intelligent chess programs.

Some of the scientists at the Dartmouth Conference predicted that within a few short years, chess programs would be able to beat even grand masters. The prediction was a little ambitious. It wasn't until 1967 that the first computer was entered into a chess tournament. During the next year, the international chess master David Levy bet that no computer would beat him at chess within the next ten years. Despite much improvement in the chess playing capability of AI programs, he won his bet. He then increased his bet and continues his search for a winning chess program. Even though some programs now play at the master level, in 1984 Levy won four games in a row against a program run on the powerful Cray X-MP computer.

Chess is not the only strategy game AI researchers have experimented with. Beginning in 1947, Arthur Samuel tried to program a computer to play a masterful game of checkers. He was one of the first to create a playing program that could beat masters of the game. It wasn't until the late 1960s that his program could beat most players. Duke University claims that its checkers program is ranked tenth in the world. Hans Berliner's BKG 9.8 backgammon program, however, beat Luigi Villa four games out of five shortly after Villa was named world backgammon champion.

Game-playing programs must be carefully planned. A program cannot be written to simply examine every possible move in a game and choose the best one. In the game of tic-tac-toe, for example, the first player has the choice of nine different moves. For each of those moves eight remain. Then seven, then six, until a winner is found or there is nowhere left to move. That means there are nearly 9 factorial moves. That's over 360,000 different tic-tac-toe games for the computer to

evaluate! David Peat (1988) estimates that there are nearly 10^{120} different possible games of chess. That's 1 followed by 120 zeros! Evaluating one million moves a second, it would take the computer nearly a trillion years to select the first move!

Game-playing programs, like chess programs, use a variety of programming approaches to cut down on the amount of time needed to select a move. Some take advantage of all that is known about classic moves. Chess especially has a rich history of famous games and classic plays. Programs can access libraries of plays to select moves. Other programs examine all of the possible moves and choose the one that will best ensure a win for the computer. Some powerful programs play by learning both good moves and moves to avoid as they are playing the game. Humans must often play against these programs for some time before the programs learn enough to play a challenging game. Still other programs evaluate moves based on ideas that are key to the game, like controlling the center of the board, the value of the pieces remaining on the board, or the number of the opponent's pieces that can be attacked from a position.

Arthur Samuel's checkers program uses this last approach. His final program took into account over 40 factors like those described above. Some AI programs use routines that mimic the procedures humans would use. Other AI programs use routines that are unnatural for a human but efficient for a machine. Samuel tried the human approach, creating a program using the same process to decide on a move that an expert checker player would use. Just discovering the human process is an enormous amount of work. Samuels interviewed checkers masters to see how they thought they selected moves. He learned quickly that experts don't really know how they do what they do. Often, masters would say that they would chose a move a certain way but observation proved that it was not the actual process used. Computer scientists who design modern expert systems still have the same problem getting information from human experts. Through persistence and careful observation of the experts, Samuel was able to get the information he needed to create his game.

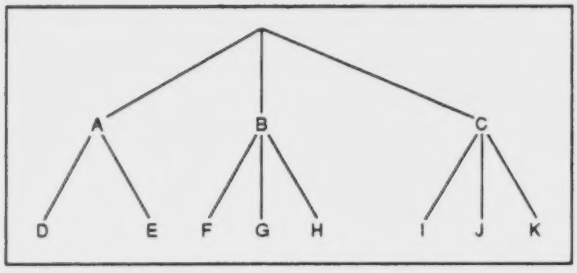
Most game-playing programs have three parts: an evaluation procedure, a minimaxing procedure, and an alpha-beta algorithm. The evaluation procedure awards points, arbitrarily assigned by the programmer, to each potential move based on the move's merits. A "good" move would receive a high score and a "bad" move would get a low score. A chess move that captures the queen would receive more points than a move that captures some lesser piece. A move that leaves a piece vulnerable to capture would possibly receive negative points.

The score for a move is based on the sum of the points for all the positive and negative facets of the move. A move that captures an opponent's piece, sets up the capture of another piece, but leaves itself open for capture would receive the points from all three factors. If too many factors are considered, the program can be frustratingly slow. Samuel saved time running his checkers program by only considering those of the 40 factors that applied to a particular move.

It is not enough to evaluate just the moves available to the computer. At first glance the move producing the highest score may seem to be the best move. How many times, however, have you played chess or checkers and thought you had just made a terrific move, only to have

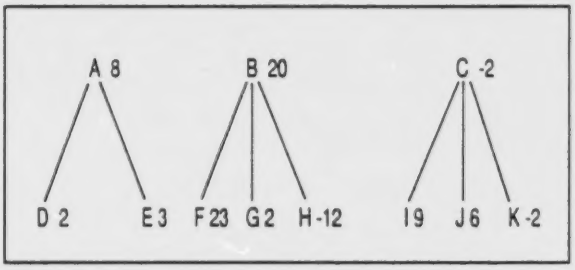
your opponent make a move you hadn't even noticed that wins the game? Capturing the queen may be tempting, but is not helpful if a pawn puts you in checkmate in the next move. Thoroughly examining and evaluating the moves available to the opponent can avoid this kind of humiliating defeat.

Consider the example where the computer, playing some hypothetical game, has a choice of three moves. After one of these moves, the human has two possible moves. After another computer move, the human has a choice of three, and after the third computer move, the human has three choices. The situation can be pictured as below. Each letter indicates a possible move for either the computer or the human. A, B, and C represent the legal moves the computer can make, and the others are those available to the opponent.



SMARTTT DRAWING 1

This type of diagram is called a tree. It even looks a little like an upside-down tree. Each line is called a branch, and those moves at the very end of each branch are called leaves. Trees are used in a wide variety of computer applications, usually those that involve searching. Each of the letters is called a node. The nodes in this example are the possible moves. In a game-playing program, each node in the tree of possible moves is evaluated and a number assigned to each node. After the hypothetical evaluation procedure is run, our tree may look like this:



SMARTTT DRAWING 2

You can see that if the computer only considered the scores of its own possible moves, it would select move B. That might, however, give the human move F, which is valued more than B. A minimaxing routine looks through the tree for the move which will give the computer the highest possible score and will give the human the lowest score, thus "minimaxing."

The program in our example would choose move A because this gives the largest difference between the best score for the human and the computer's score. Move A puts the computer ahead by five points. B puts the computer behind by three and C has the computer behind by 11! These points aren't scores for the game but rather a way the program can compare the relative strategic strength of the computer's and human's moves.

Generally, a good computer game program looks ahead more than just two moves. This can use up an enormous amount of computer memory and running time. In our example, the computer looks only two moves ahead, and already it must evaluate 11 different moves.

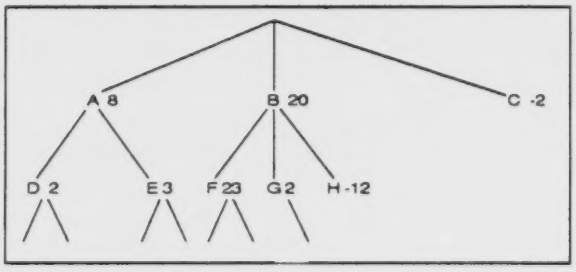
There must be ways to "prune" a strategy tree if a game played with a computer is to be completed in a reasonable amount of time. One method is to limit the depth of a search, that is, to limit how many

turns ahead the computer evaluates possible moves. If the computer only looks four turns ahead, it examines two sets of possible computer moves and two sets of possible human moves. This is called a four-ply search. If the program only looked two turns ahead, it is called a two-ply search. Even a four-ply search can still leave the computer with an astronomical number of moves to evaluate. If thirty is the average number of moves available to a chess player in any one turn, a four-ply search would mean evaluating 30 x 30 x 30 x 30 or 810,000 different moves! Another method called the alpha-beta routine can help prune a strategy tree further.

The premise of the alpha-beta routine is really quite simple: players don't make really bad moves. If a move gets a very low score after evaluation, the computer does not bother to evaluate the new set of moves that would be made possible by that move. As an example, assume once again that you are playing chess. You have several legal moves to choose from. One of those moves is to move your pawn ahead one space. You realize, however, that doing so would allow your opponent to capture your queen and put you in check. You would not think about how you would get out of check, because you would decide not to move your pawn. There is no point in looking ahead any more turns.

Alpha refers to the evaluation of a move for the computer. Beta is the evaluation of a move for the human. So in the tree above, B has an alpha value of 20 and D has a beta value of 2. The values are arrived at by the evaluation routine. The minimaxing procedure tries to find the highest alpha value with the lowest beta value. The alpha-beta algorithm helps keep that search manageable by not bothering to examine the branches from a bad move.

The computer starts by evaluating its first three possible moves. After evaluation, the program notes by the negative score that move C is not a good choice. There is no reason for the computer to select that move, so when it comes time to evaluate the human moves, the computer will not bother to evaluate moves I, J, or K as it did when using just the minimaxing procedure. The alpha-beta principal also holds that the opponent will not make a bad move either. Thus if we were looking more than just two moves ahead, the computer would not continue the evaluation of branch H. This is what our tree would look like after pruning.



SMARTTT DRAWING 3

Without the alpha-beta routine, the computer must evaluate 11 moves. With the alpha-beta routine, only eight moves are evaluated. The two-ply search is cut by a fourth with the use of the alpha-beta routine! More time would have been saved if the program evaluated four- or six-ply searches. David Levy estimates that a chess program that looks four moves ahead using the alpha-beta algorithm will run 500 times faster than a program without the alpha-beta algorithm (Krutch, 1986).

The Program

The Logo program for this chapter is called SMARTTT. It plays tic-tac-toe intelligently. Although it has an evaluation procedure, it uses a simple one-level search to try to find the best moves.

A couple of key variables keep track of the game for us. :BOARD maintains a picture of the game board. This is what the tic-tac-toe board looks like when printed by the program. Below that is the value of :BOARD before the game begins:

```
1 | 2 | 3
4 | 5 | 6
7 | 8 | 9

:BOARD is [ 1 2 3 4 5 6 7 8 9 ]
```

As the game is played, the numbers in the list :BOARD will be replaced with the letters X and O as the user and the computer start choosing squares. Procedures can be written to check for a winner and evaluate moves for the computer using just :BOARD, but procedures that run faster and are less complex can be written if we use a few other variables to keep track of things. :H.SQUARES remembers which squares the person has selected and :C.SQUARES maintains a list of the computer's positions. The squares that have not yet been used are kept in the list :REMAINING.SQRS. Below is an example of what value each variable would have given this game board:

```
1 | O | X
X | O | 6
7 | X | 9

:BOARD is [ 1 O X X O 6 7 X 9 ]
:H.SQUARES is [ 3 4 8 ]
:C.SQUARES is [ 2 5 ]
:REMAINING.SQRS is [ 1 6 7 9 ]
```

As the game is played, both :BOARD and :REMAINING.SQRS have to be modified: :BOARD must have numbers replaced by X or O, and :REMAINING.SQRS must have numbers removed from its list. Removing and replacing items in a list are fairly common routines, so the program uses some tool procedures. These procedures are written in a general way so that they can be used easily in other programs as well. You may wish to save separately the procedures REMOVED and REPLACE.ITEM to use in your own future programs.

Both have similar formats. They use recursion to look through the appropriate list for the element to be exchanged or deleted. Each considers three cases. If the list being modified is empty, the empty list is returned. If the first item in the list is the item that is to be replaced or removed, it is substituted or discarded, and the rest of the list is outputted. The third case is the general case. The first item in the list is outputted, and the procedure is called again on the rest of the list (see Weston, 1985, pp. 84-90, for an excellent detailed discussion of how these procedures work).

```
TO REPLACE.ITEM :OLD.ITEM :NEW.ITEM
  :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF EQUAL? FIRST :LIST :OLD.ITEM [OUTPUT
  FPUT :NEW.ITEM BUTFIRST :LIST]
OUTPUT FPUT FIRST :LIST REPLACE.ITEM
  :OLD.ITEM :NEW.ITEM BUTFIRST :LIST
END

TO REMOVED :ITEM :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF EQUAL? FIRST :LIST :ITEM [OUTPUT
  BUTFIRST :LIST]
OUTPUT FPUT FIRST :LIST REPLACE.ITEM
  :OLD.ITEM :NEW.ITEM BUTFIRST :LIST
END
```

Most of the procedures in SMARTTT are used to manage the game (keeping track of whose turn it is, checking to see if a player has won, etc.). The "intelligent" part of the program is the procedures that help the computer select a move. The most important of those is the evaluation procedure: EVAL.ALPHA. Recall that "alpha" refers to the value of each possible move for the computer.

When evaluating a move, EVAL.ALPHA considers four factors (hopefully you can think of more and add them to the program). First, it looks to see if the possible move would make the computer a winner by getting three in a row. If so, this move earns the highest possible number of points. Next, the procedure looks to see if the move is one of the corner squares, since they offer the most potential to win early in the game. The corner squares are 1, 3, 7, and 9, and increase the value of a move. The value of a move can also increase if it is the center square. Lastly, the procedure checks to see if the move will block the user. It does this by temporarily adding the move to the list of the human user's squares and looking to see if the combination produces a winner. Although the factors are not evaluated in order of their importance, their points, which are added to a move's total score, reflect their respective strategic value.

```
TO EVAL.ALPHA :MOVE
MAKE "VALUE 0
IF WINNER? SENTENCE :C.SQUARES :MOVE
  [OUTPUT 128 STOP]
IF MEMBER? :MOVE [1 3 7 9] [MAKE "VALUE
  :VALUE + 10]
IF :MOVE = 5 [MAKE "VALUE :VALUE + 5]
IF WINNER? SENTENCE :H.SQUARES :MOVE
  [MAKE "VALUE :VALUE + 25]
OUTPUT :VALUE
END
```

COMPUTER.MOVE controls the selection of the program's square. This procedure looks for the best move, makes it, and tells the user which square it selected. The procedure that actually looks for the best move is FIND.BEST.MOVE. This is an initialization procedure; it sets things up for the next procedure called. The variable :ALPHA is a list made up of the best move found so far and the alpha value of that move. For each of the remaining squares, EVAL.MOVES calls EVAL.ALPHA to find the alpha value for that move. If its alpha value is higher than the current one, :ALPHA is updated with the new best move and its score. EVAL.MOVES keeps the user posted as to which move it has just considered, its score, and the current alpha (best score).

```
TO COMPUTER.MOVE
PRINTBOARD
MAKE "TEMP.SQRS :REMAINING.SQRS
FIND.BEST.MOVE
MAKEMOVE "O
MAKE "C.SQUARES SENTENCE :C.SQUARES
  :MOVE
MAKE "REMAINING.SQRS REMOVED :MOVE
  :REMAINING.SQRS
PR []
(PR [I'LL TAKE:] :MOVE)
PR []
PRINTBOARD
END

TO FIND.BEST.MOVE
IF (COUNT :REMAINING.SQRS) = 1 [MAKE
  "MOVE FIRST :REMAINING.SQRS STOP]
MAKE "ALPHA [-9 -9]
EVAL.MOVES :REMAINING.SQRS
MAKE "MOVE FIRST :ALPHA
END
```

```
TO EVAL.MOVES :TEMP.SQRS
IF (COUNT :TEMP.SQRS) < 1 [STOP]
MAKE "SCORE EVAL.ALPHA FIRST :TEMP.SQRS
IF (LAST :ALPHA) < :SCORE [MAKE "ALPHA
  SENTENCE FIRST :TEMP.SQRS :SCORE]
(PRINT [CONSIDERING:] FIRST :TEMP.SQRS
  [SCORE:]:SCORE [ALPHA:] LAST :ALPHA)
EVAL.MOVES BUTFIRST :TEMP.SQRS
END
```

The reporter LEGAL? is a predicate that outputs true if the item in the square chosen is still a number. If the item in the square were the letter X or O, then the square would already be taken.

Two procedures are used in tandem to determine if there is a winner. These are the reporters WINNER? and SUB.WIN. WINNER? takes care of initialization of :RESULT and :TEMP.LIST and then calls SUB.WIN which recursively looks for a possible win. The procedure does this by looking one-by-one at the possible ways to win to see if the user or the computer has in their list of squares those three positions. In effect, the procedure looks to see if either the computer or the user has three in a row. After SUB.WIN checks one possible way to win, it moves on to the next possibility.

```
TO LEGAL?
OUTPUT NUMBER? ITEM :MOVE :BOARD
END

TO WINNER? :PLAYER
IF (COUNT :PLAYER) < 3 [OUTPUT "FALSE
  STOP]
OUTPUT SUB.WIN :POSS.WINS
END

TO SUB.WIN :TEMP.LIST
IF :TEMP.LIST = [] [OUTPUT "FALSE]
MAKE "TEST.SET FIRST :TEMP.LIST
IF (AND MEMBER? FIRST :TEST.SET :PLAYER
  MEMBER? FIRST BUTFIRST :TEST.SET
  :PLAYER MEMBER? LAST :TEST.SET
  :PLAYER)
  [OUTPUT "TRUE]
SUB.WIN BUTFIRST :TEMP.LIST
END
```

HUMAN.MOVE is the procedure that handles the person's selection of a square. The user selects a move by typing the number of the square he or she wishes to select. HUMAN.MOVE accepts that move as input, checks to make sure it is legal, and makes the move. The procedure also adds the move to the user's list of squares and deletes the square from the list of those available.

```
TO HUMAN.MOVE
CC
TYPE [WHAT IS YOUR MOVE?]
MAKE "MOVE FIRST READLISTCC
IFELSE LEGAL? [MAKEMOVE "X MAKE
  "H.SQUARES SENTENCE :H.SQUARES :MOVE
  MAKE "REMAINING.SQRS REMOVED :MOVE
  :REMAINING.SQRS]
  [PRINT [ILLEGAL MOVE!] HUMAN.MOVE]
CT
END
```

A handful of procedures control the operation of the entire program. SMARTTT is the superprocedure. The procedure INITIALIZE sets up some of the key variables. PRINTBOARD displays the state of the board, showing the user's squares with X, the computer's moves with O, and the remaining squares with the corresponding number. The main portion of the program is TTT.LOOP. This procedure facilitates the cycle of getting the computer's and person's moves and then checks for a win after each. The user always goes first and uses X to represent completed moves. The computer uses O. Since the person goes first, a draw can occur only after the user's move, thus the single check for a draw after the human's move in TTT.LOOP.

```
TO SMARTTT
INITIALIZE
PRINTBOARD
TTT.LOOP
END
```

```
TO  INITIALIZE
MAKE "POSS.WINS [ [1 2 3] [4 5 6]
   [7 8 9] [1 4 7] [2 5 8] [3 6 9]
   [1 5 9] [7 5 3] ]
MAKE "BOARD [1 2 3 4 5 6 7 8 9]
MAKE "H.SQUARES []
MAKE "C.SQUARES []
MAKE "REMAINING.SQRS :BOARD
CT HT
END

TO  TTT.LOOP
HUMAN.MOVE
IF WINNER? :H.SQUARES [HUMAN.WINS STOP]
IF :REMAINING.SQRS = [] [DRAW.GAME
  STOP]
COMPUTER.MOVE
IF WINNER? :C.SQUARES [COMP.WINS STOP]
TTT.LOOP
END

TO  PRINTBOARD
PR []
(PR ITEM 1 :BOARD ": ITEM 2 :BOARD ":
  ITEM 3 :BOARD)
PR "_________
(PR ITEM 4 :BOARD ": ITEM 5 :BOARD ":
  ITEM 6 :BOARD)
PR "_________
(PR ITEM 7 :BOARD ": ITEM 8 :BOARD ":
  ITEM 9 :BOARD)
PR []
END
```

Possible Extensions and Expansions

- Use graphics to create the board.
- Have the computer remember all the moves with the highest score and then select randomly from the list. Right now, the program only remembers the first move of highest value. For example, in beginning play, the corners have the highest strategic value. Even if the person selects a corner square, there are three other corners for the computer to choose from. The program will take square 1 (3 if the person took 1) because it is the first corner it considered, but it would be better if the computer selected randomly from each of the available corners.
- Expand and reorganize the program so that either the computer or the person can go first. One approach would be to have the program randomly select who went first or each could take turns going first. Don't forget to check for a draw after each move.
- If a game ends in a draw, it might be nice to wipe the board clean and continue the game. This expansion requires that either the computer or the person be able to make the first move.
- Add to the list of factors the computer considers when evaluating a move. You might get ideas of other factors or strategies for winning by playing the game with a friend for a while or by interviewing and observing others while they play the game.
- One possible factor would be to have the computer not select a square if there is no way the computer could win from that square. For example (recall that the computer is O), in the board configuration below, the computer should not select 3 or 9 because it could not win from those squares. What should the value of such a square be relative to the other factors? That is, what factors should receive higher values so that the computer will select a square from which they cannot win? You may have to change the scores of some of the existing factors.

```
X | 2 | 3
O | O | X
X | 8 | 9
```

- Have the computer determine when there will be no winner as far ahead as possible.
- Implement the beta phase of finding a move for the computer. This would require modifying FIND.BEST.MOVE and the procedures which help it, to look at the person's possible moves for each possible computer move. A minimax routine might subtract the highest value for a person's move from the value of the computer's move. These net scores could then be used to select the best move.
- If you complete the above suggestion, further modify the program so that it uses the alpha-beta algorithm (doesn't continue considering a bad move).
- Extend the program even further to look more than two moves ahead.
- Try to create a game like "Connect Four." It uses an eight-by-eight board, and the winner is the first to get four in a row. "Connect Four" is played vertically, so for a move to be legal, the move must be in the bottom row or on top of an existing piece. It may be a lot easier just to modify the rules so that any square can be selected as in tic-tac-toe.

SMARTTT
THE PROGRAM
LogoWriter Version

```
TO  SMARTTT
INITIALIZE
PRINTBOARD
TTT.LOOP
END

TO  INITIALIZE
MAKE "POSS.WINS [ [1 2 3] [4 5 6]
   [7 8 9] [1 4 7] [2 5 8] [3 6 9]
   [1 5 9] [7 5 3] ]
MAKE "BOARD [1 2 3 4 5 6 7 8 9]
MAKE  "H.SQUARES  []
MAKE  "C.SQUARES  []
MAKE  "REMAINING.SQRS  :BOARD
CT HT
END

TO  TTT.LOOP
HUMAN.MOVE
IF  WINNER?  :H.SQUARES  [HUMAN.WINS  STOP]
IF  :REMAINING.SQRS  =  []  [DRAW.GAME
   STOP]
COMPUTER.MOVE
IF  WINNER?  :C.SQUARES  [COMP.WINS  STOP]
TTT.LOOP
END

TO  PRINTBOARD
CT
PR []
(PR ITEM 1 :BOARD ":  ITEM 2 :BOARD ":
   ITEM 3 :BOARD)
PR  "_________
(PR ITEM 4 :BOARD ":  ITEM 5 :BOARD ":
   ITEM 6 :BOARD)
PR  "_________
(PR ITEM 7 :BOARD ":  ITEM 8 :BOARD ":
   ITEM 9 :BOARD)
PR []
END
```

```
TO HUMAN.MOVE
CC
TYPE [WHAT IS YOUR MOVE?]
MAKE "MOVE FIRST READLISTCC
  IFELSE LEGAL? [MAKEMOVE "X MAKE
  "H.SQUARES SENTENCE :H.SQUARES :MOVE
  MAKE "REMAINING.SQRS REMOVED :MOVE
  :REMAINING.SQRS]
  [PRINT [ILLEGAL MOVE!] HUMAN.MOVE]
CT
END

TO WINNER? :PLAYER
IF (COUNT :PLAYER) < 3 [OUTPUT "FALSE]
OUTPUT SUB.WIN :POSS.WINS
END

TO SUB.WIN :TEMP.LIST
IF :TEMP.LIST = [] [OUTPUT "FALSE]
MAKE "TEST.SET FIRST :TEMP.LIST
IF (AND MEMBER? FIRST :TEST.SET :PLAYER
  MEMBER? FIRST BUTFIRST :TEST.SET
  PLAYER MEMBER? LAST :TEST.SET
  PLAYER)
  [OUTPUT "TRUE]
OUTPUT SUB.WIN BUTFIRST :TEMP.LIST
END

TO HUMAN.WINS
PRINTBOARD
PR []
PR [YOU WON!]
PR []
PLAY.AGAIN
END

TO COMPUTER.MOVE
PRINTBOARD
MAKE "TEMP.SQRS :REMAINING.SQRS
FIND.BEST.MOVE
MAKEMOVE "O
MAKE "C.SQUARES SENTENCE :C.SQUARES
  :MOVE
MAKE "REMAINING.SQRS REMOVED :MOVE
  :REMAINING.SQRS
PR []
(PR [I'LL TAKE:] :MOVE)
PR []
PRINTBOARD
END

TO COMP.WINS
PRINTBOARD
PR []
PR [I WON!]
PR []
PLAY.AGAIN
END

TO DRAW.GAME
PRINTBOARD
PR []
PR [IT'S A DRAW!]
PR []
PLAY.AGAIN
END

TO LEGAL?
OUTPUT NUMBER? ITEM :MOVE :BOARD
END

TO MAKEMOVE :XO
MAKE "BOARD REPLACE.ITEM :MOVE :XO
  :BOARD
END

TO REPLACE.ITEM :OLD.ITEM :NEW.ITEM
  :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF EQUAL? FIRST :LIST :OLD.ITEM [OUTPUT
  FPUT :NEW.ITEM BUTFIRST :LIST]
OUTPUT FPUT FIRST :LIST REPLACE.ITEM
  :OLD.ITEM :NEW.ITEM BUTFIRST :LIST
END

TO REMOVED :ITEM :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF EQUAL? FIRST :LIST :ITEM [OUTPUT
  BUTFIRST :LIST]
OUTPUT FPUT FIRST :LIST REMOVED ITEM
  BUTFIRST :LIST
END

TO PLAY.AGAIN
PR [DO YOU WANT TO PLAY AGAIN?]
MAKE "ANS FIRST READLIST
IF (FIRST :ANS) = "Y [SMARTTT]
PR []
PR [GOODBYE]
REPEAT 4 [PR []]
END
```

```
TO  FIND.BEST.MOVE
IF  (COUNT  :REMAINING.SQRS)  =  1  [MAKE
   "MOVE  FIRST  :REMAINING.SQRS  STOP]
MAKE  "ALPHA  [-9  -9]
EVAL.MOVES  :REMAINING.SQRS
MAKE  "MOVE  FIRST  :ALPHA
END

TO  EVAL.MOVES  :TEMP.SQRS
IF  (COUNT  :TEMP.SQRS)  <  1  [STOP]
MAKE  "SCORE  EVAL.ALPHA  FIRST  :TEMP.SQRS
IF  (LAST  :ALPHA)  <  :SCORE  [MAKE  "ALPHA
   SENTENCE  FIRST  :TEMP.SQRS  :SCORE]
(PRINT  [CONSIDERING:]  FIRST  :TEMP.SQRS
   [SCORE:]  :SCORE
   [ALPHA:]  LAST  :ALPHA)
EVAL.MOVES  BUTFIRST  :TEMP.SQRS
END

TO  EVAL.ALPHA  :MOVE
MAKE  "VALUE  0
IF  WINNER?  SENTENCE  :C.SQUARES  :MOVE
   [OUTPUT  128  STOP]
IF  MEMBER?  :MOVE  [1  3  7  9]  [MAKE  "VALUE
   :VALUE  +  10]
IF  :MOVE  =  5  [MAKE  "VALUE  :VALUE  +  5]
IF  WINNER?  SENTENCE  :H.SQUARES  :MOVE
   [MAKE  "VALUE  :VALUE  +  25]
OUTPUT  :VALUE
END
```

SMARTTT
THE PROGRAM
Terrapin Logo PLUS Version

```
TO  SMARTTT
  INITIALIZE
  TTT.LOOP
END

TO  INITIALIZE
  MAKE  "POSS.WINS  [  [1  2  3]  [4  5  6]
   [7  8  9]  [1  4  7]  [2  5  8]  [3  6  9]
   [1  5  9]  [7  5  3]  ]
  MAKE  "BOARD  [1  2  3  4  5  6  7  8  9]
  MAKE  "H.SQUARES  []
  MAKE  "C.SQUARES  []
  MAKE  "REMAINING.SQRS  :BOARD
  CLEARTEXT
END

TO  TTT.LOOP
  PRINTBOARD
  HUMAN.MOVE
  IF  WINNER?  :H.SQUARES  THEN  HUMAN.WINS
    STOP
  IF  :REMAINING.SQRS  =  []  THEN
    DRAW.GAME  STOP
  PRINTBOARD
  COMPUTER.MOVE
  IF  WINNER?  :C.SQUARES  THEN  COMP.WINS
    STOP
  TTT.LOOP
END

TO  PRINTBOARD
  PR  []
  PR  (SENTENCE  ITEM  1  :BOARD  "|  ITEM  2
    :BOARD  "|  ITEM  3  :BOARD)
  PR  "_________
  PR  (SENTENCE  ITEM  4  :BOARD  "|  ITEM  5
    :BOARD  "|  ITEM  6  :BOARD)
  PR  "_________
  PR  (SENTENCE  ITEM  7  :BOARD  "|  ITEM  8
    :BOARD  "|  ITEM  9  :BOARD)
  PR  []
END
```

```
TO HUMAN.MOVE
 PRINT1 ['WHAT IS YOUR MOVE? ']
 MAKE "MOVE FIRST REQUEST
 TEST LEGAL?
 IFTRUE MAKEMOVE "X
 IFTRUE MAKE "H.SQUARES SENTENCE
  :H.SQUARES :MOVE
 IFTRUE MAKE "REMAINING.SQRS REMOVED
  :MOVE :REMAINING.SQRS
 IFFALSE PRINT [ILLEGAL MOVE!]
  HUMAN.MOVE
END

TO WINNER? :PLAYER
  IF COUNT :PLAYER < 3 THEN OUTPUT
  "FALSE
  OUTPUT SUB.WIN :POSS.WINS
END

TO SUB.WIN: TEMP.LIST
 IF :TEMP.LIST = [] THEN OUTPUT "FALSE
 MAKE "TEST.SET FIRST :TEMP.LIST
 IF (ALLOF MEMBER? FIRST :TEST.SET
  :PLAYER
  MEMBER? FIRST BUTFIRST :TEST.SET
  :PLAYER MEMBER? LAST :TEST.SET
  :PLAYER)
  THEN OUTPUT "TRUE
 OUTPUT SUB.WIN BUTFIRST :TEMP.LIST
END

TO HUMAN.WINS
 PRINTBOARD
 PR []
 PR [YOU WON!]
 PR []
 PLAY.AGAIN
END

TO COMPUTER.MOVE
 FIND.BEST.MOVE
 MAKEMOVE "O
 MAKE "C.SQUARES SENTENCE :C.SQUARES
  :MOVE
 MAKE "REMAINING.SQRS REMOVED :MOVE
  :REMAINING.SQRS
 PR []
 PR SENTENCE [I'LL TAKE] :MOVE
 PR []
END

TO COMP.WINS
 PRINTBOARD
 PR []
 PR [I WON!]
 PR []
 PLAY.AGAIN
END

TO DRAW.GAME
 PRINTBOARD
 PR []
 PR [IT'S A DRAW]
 PR []
 PLAY.AGAIN
END

TO LEGAL?
 OUTPUT NUMBER? ITEM :MOVE :BOARD
END

TO MAKEMOVE :XO
 MAKE "BOARD REPLACE :MOVE :XO :BOARD
END

TO REPLACE :OLD.ITEM :NEW.ITEM :LIST
 IF EMPTY? :LIST THEN OUTPUT :LIST
 IF EQUAL? FIRST :LIST :OLD.ITEM THEN
 OUTPUT FPUT :NEW.ITEM BUTFIRST :LIST
 OUTPUT FPUT FIRST :LIST REPLACE
  :OLD.ITEM :NEW.ITEM BUTFIRST :LIST
END

TO REMOVED :ITEM :LIST
 IF EMPTY? :LIST THEN OUTPUT :LIST
 IF EQUAL? FIRST :LIST :ITEM THEN
  OUTPUT BUTFIRST :LIST
 OUTPUT FPUT FIRST :LIST REMOVED :ITEM
  BUTFIRST :LIST
END

TO EQUAL? :ITEM1 :ITEM2
 OUTPUT :ITEM1 = :ITEM2
END


TO PLAY.AGAIN
 PR [DO YOU WANT TO PLAY AGAIN?]
 MAKE "ANS FIRST REQUEST
 IF FIRST :ANS = "Y THEN SMARTTT
 PR []
 PR [GOODBYE]
 REPEAT 4 [PR []]
END
```

```
TO  FIND.BEST.MOVE
  IF  COUNT  :REMAINING.SQRS  =  1  THEN  MAKE
    "MOVE  FIRST  :REMAINING.SQRS
  MAKE  "TEST.MOVE  FIRST  :REMAINING.SQRS
  PRINT1  SENTENCE  [CONSIDERING]
    :TEST.MOVE
  MAKE  "SCORE  EVAL.ALPHA  :TEST.MOVE
  MAKE  "ALPHA  SENTENCE  :TEST.MOVE  :SCORE
  PR  (SENTENCE  [' SCORE:']  :SCORE
    [ALPHA:]  LAST  :ALPHA)
  MAKE  "TEMP.SQRS  BUTFIRST
    :REMAINING.SQRS
  EVAL.MOVES
  MAKE  "MOVE  FIRST  :ALPHA
END

TO  EVAL.MOVES
  IF  :TEMP.SQRS  =  []  THEN  STOP
  PRINT1  SENTENCE  [CONSIDERING]  FIRST
    :TEMP.SQRS
  MAKE  "SCORE  EVAL.ALPHA  FIRST
    :TEMP.SQRS
  PR  (SENTENCE  [' SCORE:']  :SCORE
    [ALPHA:]  LAST  :ALPHA)
  IF  LAST  :ALPHA  <  :SCORE  THEN  MAKE
    "ALPHA  SENTENCE  FIRST  :TEMP.SQRS
    :SCORE
  MAKE  "TEMP.SQRS  BUTFIRST  :TEMP.SQRS
  EVAL.MOVES
END

TO  EVAL.ALPHA  :MOVE
  MAKE  "VALUE  0
  IF  WINNER?  SENTENCE  :C.SQUARES  :MOVE
    THEN  OUTPUT  128  STOP
  IF  MEMBER?  :MOVE  [1  3  7  9]  THEN  MAKE
    "VALUE  :VALUE  +  10
  IF  :MOVE  =  5  THEN  MAKE  "VALUE  :VALUE  +
    5
  IF  WINNER?  SENTENCE  :H.SQUARES  :MOVE
    THEN  MAKE  "VALUE  :VALUE  +  25
  OUTPUT  :VALUE
END
```

3 TALK.AND.DRAW— LANGUAGES FOR HUMANS AND MACHINES

Several years ago, I was helping a student with her first programming assignment. She was to program the computer to find the square root of twenty-five. She sat down to the computer and at the prompt wrote, "Find the square root of 25." The computer responded with some seemingly rude comment about syntax errors. Why didn't the computer understand a perfectly normal sentence? Because computers and humans speak different languages.

There are several types of languages. A language is a set of symbols (vocabulary) together with some rules for putting the symbols together (grammar). Languages differ by the size of their vocabularies, the complexity of their grammars and their flexibility of use. Human languages, called natural languages, have rich vocabularies. The grammars are extensive, but people still usually understand the meaning of the speaker even when the rules are not followed. Human languages are special in that they can be used for everything from giving orders to discussing philosophy to writing poetry.

Computer languages, on the other hand, are very limited. Not only is the singular purpose of computer languages to direct the machine but the vocabularies and grammars are also very restricted. Computers are electronic machines, and the simplest method of "communication" for electronics is the switch. The vocabulary is simple: strings of "on"s and "off"s (usually represented as 1s and 0s). The computer understands what it should put on the screen, get from the disk, or print on the printer by reading this binary language. The grammar for machine language is composed of few rules, but they must be followed rigidly or the machine will not understand. The computer can operate very quickly when using machine language, but the language does not facilitate human understanding. Imagine trying to read and understand a page full of nothing but the digits one and zero!

Early programs written in this binary code functioned just fine but were difficult to work with. The binary codes for various commands were hard to remember and, as most programs were long, consisted of long blocks of 0s and 1s. Tired of the bleary eyes produced by reading binary code, computer scientists wrote programs (in binary) that would translate more readable codes into binary. At first, the codes took the form of simple three or four letter "words" that stood for specific binary commands. Since these new codes were often mnemonics for the actual command, they were much easier to remember (e.g., JMP for Jump). These simple translator programs are called Assemblers and the mnemonics are called Assembly Language.

Soon, programmers realized they were using the same strings of these new commands over and over. They created new translator programs that converted yet more codes into these commonly used strings of commands. These new codes were much easier for people to use. Doing something as simple as calculating a two- or three-step arithmetic problem took several lines of Machine Language or Assembly Language but took only a single line of these new codes, or high level languages. Examples of high level languages include BASIC, Pascal, and Logo.

High level languages are currently the most common solution to the communication gap between people and computers. The vocabularies are small, but the words

are similar to English and therefore much easier to remember and understand than the ones and zeros of machine language. There is, however, little flexibility to high level language grammars. In order for the translator programs to be efficient, the high level languages must themselves be simple; that means limited vocabulary and rigid syntax. The student's request to find a square root used words that were not in the computer's vocabulary, and her sentence, although using correct English grammar, did not use correct computer grammar.

The next logical step in closing the communication gap between computers and people is natural language processors. Natural language processors (NLPs) are programs that translate human languages into languages the computer can understand. NLPs will open the world of computing to even more people. If the girl I was tutoring had a computer equipped with such a processor, she could have asked the computer "What is the square root of 25?" The NLP would have translated the English request into some other language the computer could process, and the computer would have computed the result.

There are many different approaches to creating natural language processors. Most of the computer users writing such translators fall somewhere between two groups. One group writes programs to make computers easier to use. They use whatever programming methods they can to create programs that work. Their main goal is translator programs that are practical and can be used in applications to allow humans greater ease of communication with their machine. These translator programs usually focus on the limited context of some application like database searching or computation.

The other group of computer scientists create NLPs in an attempt to better understand how people process the languages they speak. These scientists are very particular about the programming methods used. They try to create programs that mirror human processes. Their programs are simulations of how people understand language. Cognitive specialists work with computer scientists. Together they make hypotheses and educated guesses of how people use language, and then write programs that fit those hypotheses. These language translation programs are more general than their limited-scope counterparts described above. The scientists who write them hope that one day there will be a general NLP that can translate any English sentence into commands a computer can process.

Both types of programmers face many of the same problems. The largest problem is that no one knows how people understand language. The one thing we know for sure is that language is a very complex system. Natural languages have enormous vocabularies and complex rules of syntax. The meaning of a sentence is often subtle and must be taken in context of a reading or conversation. Also, people understand sentences that contain words they don't understand and that use nonstandard syntax. Even if we knew the human procedures, the programs we wrote to simulate them would be very long and complex.

The complexity of the language that can be used to communicate with the computer depends heavily on the amount of memory available and the speed of the microprocessor. As computer memories grow, so does the power of the tools the computer owner can purchase.

Machine language requires little memory and can be processed very quickly. High level languages require more memory and are much slower than machine language. Because of the complexity of the languages people use, natural language is

even slower to process and requires much more memory. High level languages will continue to be used until the power of computers is great enough to support natural language processors.

The price of computer memory continues to drop. Scientific breakthroughs continue to make larger blocks of memory available. Machines with large memories need speedy processors just so they can access the memory quickly. Advances toward faster machines are also made regularly, and soon neither speed nor memory should be an obstacle to developing natural language processors.

Some more powerful microcomputers already support programs that use natural language interfaces. Although some interfaces work like high level languages that are translated directly into machine language, other natural language processors translate English into high level language commands that are then executed.

We are going to create a program that functions like commercial natural language processors. In order to create a natural language interface we need to decide what kinds of actions we wish to control with English and what kinds of sentences we might use to indicate those actions. We need to consider not only the structure of the English sentences we use but also the structure of the commands they will be translated to.

Let's limit our domain to turtle graphics commands to make our work a little easier. Our program will be able to control the Logo graphics turtle with common English. There are a lot of options. Tackling a problem like this can seem insurmountable. Where do we begin? How are we ever going to be able to interpret a sentence like "Move ahead thirty steps and then turn counter-clockwise ninety degrees"?

Let's consider the command FORWARD. Think of all the ways you might tell the turtle in English to go FORWARD. You might think of "Walk ahead 30 paces," and "Go on another hundred steps." There are certainly lots of ways. If our goal is to write a natural language processor that can understand any English sentence pertaining to turtle graphics, then we must create a translator that can process all possible sentences. If our goal is to write a natural language interface so non-programmers can easily control the graphics turtle, then we can focus on the most likely ways a person would try to tell the turtle to go forward.

The Program

This latter approach is easier and a better choice for beginning artificial intelligence programmers. Our translator will use a simple approach. It will look through each sentence in search of synonyms for Logo graphics commands. We can use this method to reduce the broad vocabulary of English into the limited vocabulary of Logo.

```
TO  PARSE.LIST  :ATOM  :INPUT.LIST
IF  :INPUT.LIST = []  [STOP]
IF  MEMBER?  :ATOM  :PROCEDURES  [MAKE
  "COMMAND  :ATOM  MAKE  "INPUT  []]
IF  :ATOM = "DRAW  [PROCEDURE.CALL
  :INPUT.LIST  MAKE  "INPUT.LIST  [END]
  STOP]
IF  MEMBER?  :ATOM  :END.LIST  [MAKE  "COMMAND
  "KWIT]
IF  MEMBER?  :ATOM  :AND.LIST  [ACT  MAKE
  "COMMAND  "  MAKE  "INPUT  []]
IF  NUMBER?  :ATOM  [MAKE  "INPUT  :ATOM  ]
IF  MEMBER?  :ATOM  :FD.LIST  [MAKE  "COMMAND
  "FORWARD  ]
IF  MEMBER?  :ATOM  :BK.LIST  [MAKE  "COMMAND
  "BACK]
IF  MEMBER?  :ATOM  :CG.LIST  [MAKE  "COMMAND
  "CG]
IF  MEMBER?  :ATOM  :RT.LIST  [MAKE  "COMMAND
  "RIGHT]
IF  MEMBER?  :ATOM  :LT.LIST  [MAKE  "COMMAND
  "LEFT]
IF  (COUNT  :INPUT.LIST)  >  1  [PARSE.LIST
  FIRST  :INPUT.LIST  BUTFIRST
  :INPUT.LIST]
END
```

```
TO ACT
IF :COMMAND = " [(TYPE [I DON'T
  UNDERSTAND] CHAR 13) STOP]
MAKE "COMMAND SENTENCE :COMMAND :INPUT
IF :DEFINING? [MAKE "ROUTINE SENTENCE
  :ROUTINE :COMMAND]
RUN :COMMAND
END
```

The procedure which does this is called PARSE.LIST. Parsing is the process of breaking a sentence into its component parts. PARSE.LIST first checks to make sure the entire sentence is not finished being processed. Then it examines the first word in the remaining part of the sentence. The procedure then runs through several checks trying to discover the appropriate command or argument the user intended. Examine the line IF MEMBER? :ATOM :FD.LIST [MAKE "COMMAND "FORWARD]. If the current word, stored in :ATOM, is one of the words meaning FORWARD, stored in FD.LIST, then the command is assumed to be FORWARD.

In addition to looking for synonyms of commands, PARSE.LIST looks for other words. If the current word is a number, it is assumed to be the input to a command. The words "and" or "then" are special, telling the parser that there are two commands in the sentence. The program parses and executes the first command and then parses and executes the second. After checking the word for each possibility, the rest of the sentence is parsed. If the current word is not known, it is simply ignored.

Once the entire sentence is checked, ACT is called. If a command has not been found, the program responds with "I don't understand" and the user can try again. Otherwise, the command and the argument are joined into one list, and then a special Logo command is invoked. If you are defining a new procedure (fully explained later), the command is added to the others stored in :ROUTINE. RUN takes the list of :COMMAND and :INPUT as its input and then executes the list as if it were a typed command line. The program is now ready to process another sentence.

If the sentence were "Move ahead 50 turtle steps then turn right 90 degrees," PARSE.LIST would recognize "move" as meaning FORWARD. "Ahead" is also recognized as meaning FORWARD. "50" becomes the input since it is a number. "Turtle" and "steps" are ignored because the program does not recognize either word. The word "then" is noticed and ACT is called, executing the command "FORWARD 50." The rest of the sentence is then parsed. "Turn" and "degrees" are ignored but "right" and "90" are identified as the command and input.

Notice that this system works even when there are some contradictions. Look at the sentence "Move back 25 steps." The intent is to invoke the command BACK but this sentence contains synonyms for both FORWARD and BACK. An unplanned benefit of the procedure we are using is that the words in the sentence are processed in their order and ACT is not called until the entire sentence has been examined. This means that although the command becomes FORWARD when PARSE.LIST looks at the first word, the command is changed to BACK when the second word is examined!

PARSE.LIST needs some support procedures before TALK.AND.DRAW can start its translating. These must initialize the lists of command synonyms. This is done in the procedure SETUP along with the initialization of other variables.

```
TO SETUP
MAKE "DEFINING? "FALSE
MAKE "FD.LIST [FORWARD AHEAD ADVANCE FD
  GO MOVE]
MAKE "BK.LIST [BACK BACKUP BACKWARDS
  RETREAT BK]
MAKE "END.LIST [END QUIT GOODBYE STOP
  TERMINATE]
MAKE "CG.LIST [CLEAR CLEAN CG]
MAKE "RT.LIST [RIGHT RT CLOCKWISE]
MAKE "LT.LIST [LEFT LT COUNTER-CLOCKWISE]
MAKE "AND.LIST [AND THEN]
MAKE "ARTICLES [A AN THE]
MAKE "PROCEDURES []
MAKE "COMMAND [CG]
CC
CT
CG
END
```

Notice that included in the synonym lists are the standard Logo commands and abbreviations. This enables the program to process "BACK 30" as easily as it does "Back up 30 turtle steps" or "LT 90" with as little effort as "Pivot 90 degrees to the left."

Now that initialization is complete, TD.LOOP is called. This procedure handles the routine that is repeated over and over: getting the English sentence, translating it into Logo commands, and then executing those commands. INIT gets the English sentence; the translating is done by PARSE.LIST and the execution by ACT.

```
TO TD.LOOP
IF :COMMAND = [QUIT] [STOP]
INIT
PARSE.LIST FIRST :ENGLISH :ENGLISH
ACT
TD.LOOP
END

TO INIT
MAKE "COMMAND "
MAKE "INPUT []
TYPE ["=>]
MAKE "ENGLISH READLISTCC
END
```

TALK.AND.DRAW also allows the user to define procedures. Once a procedure has been defined, its name is added to a list of known procedures. If it is called again, it will be run just as if it were a Logo primitive. The program knows that the user wants to define a procedure when the sentence begins with the word "draw." When PARSE.LIST sees the word "draw," the procedure PROCEDURE.CALL is invoked to do three things. First it figures out the name of the procedure. Next, it sees if the procedure is already known. If it is, it is run; but if the routine is not known, the program learns how to do it.

A typical sentence to begin defining a procedure would be "draw a square." The first part of the procedure looks at this sentence and sees that after the word "draw" is an article. The first two words of the sentence are stripped away and the procedure to be defined is titled "square." It should be noted that TALK.AND.DRAW would not work as planned if two words were left. For example, if the original sentence were "draw a big square," after parsing, the remaining title of the procedure should be "big square" but the program only handles single word titles. The procedure title would be "big."

```
TO PROCEDURE.CALL :INPUT.LIST
IFELSE MEMBER? FIRST BUTFIRST :INPUT.LIST
  :ARTICLES
  [MAKE "PROCEDURE FIRST BUTFIRST
  BUTFIRST :INPUT.LIST]
  [MAKE "PROCEDURE FIRST BUTFIRST
  :INPUT.LIST]
IF MEMBER? :PROCEDURE :PROCEDURES [MAKE
   "COMMAND :PROCEDURE
   MAKE "INPUT [] STOP]
( TYPE [TELL ME HOW TO DRAW A] :PROCEDURE
  CHAR 13)
MAKE "DEFINING? "TRUE
MAKE "DONE "FALSE
MAKE "ROUTINE []
PROC.LOOP
MAKE "PROCEDURES SENTENCE :PROCEDURES
  :PROCEDURE
```

```
DEFINE  :PROCEDURE  :ROUTINE
(TYPE  :PROCEDURE  [IS  NOW  DEFINED]
  CHAR  13)
MAKE  "DEFINING?  "FALSE
END

TO  PROC.LOOP
INIT
PARSE.PROC  FIRST  :ENGLISH  :ENGLISH
ACT
IF  NOT  :DONE  [PROC.LOOP]
END
```

Back to the first example. PROCEDURE. CALL looks to see if "square" has already been defined. Seeing that it has not, the program asks how to draw one and dives into a routine to learn and record each step. This is handled by PROC.LOOP, which is almost identical to TD.LOOP. The major change is that PARSE.PROC is called instead of PARSE.LIST. These two procedures are also similar; in PARSE.PROC, only predefined procedures may be called and if a word meaning "end" is used, it means the end of defining the procedure.

```
TO  PARSE.PROC  :ATOM  :INPUT.LIST
IF  :INPUT.LIST  =  []  [STOP]
IF  MEMBER?  :ATOM  :END.LIST  [MAKE  "COMMAND
  "STOP  MAKE  "DONE  "TRUE  STOP]
IF  MEMBER?  :ATOM  :PROCEDURES  [MAKE
  "COMMAND  :ATOM  MAKE  "INPUT  []]
IF  MEMBER?  :ATOM  :AND.LIST  [ACT  MAKE
  "COMMAND  "  MAKE  "INPUT  []]
IF  NUMBER?  :ATOM  [MAKE  "INPUT  :ATOM]
IF  MEMBER?  :ATOM  :FD.LIST  [MAKE  "COMMAND
  "FORWARD]
IF  MEMBER?  :ATOM  :BK.LIST  [MAKE  "COMMAND
  "BACK]
IF  MEMBER?  :ATOM  :CG.LIST  [MAKE  "COMMAND
  "CG]
IF  MEMBER?  :ATOM  :RT.LIST  [MAKE  "COMMAND
  "RIGHT]
IF  MEMBER?  :ATOM  :LT.LIST  [MAKE  "COMMAND
  "LEFT]
IF  NOT  :DONE  [PARSE.PROC  FIRST
  :INPUT.LIST  BUTFIRST  :INPUT.LIST]
END
```

```
TO  DEFINE  :NAME  :COMMANDS
IF  FRONT?  [FLIP]
BOTTOM
PRINT  []
PRINT  SENTENCE  "TO  :NAME
PRINT  :COMMANDS
PRINT  [END]
FLIP
END
```

The same routine ACT is used by PROC.LOOP as was used by TD.LOOP. When a procedure is being created, the variable :DEFINING? is set to TRUE and this causes ACT to record each command in :ROUTINE. After all of the commands are stored as a list of lists, they are defined as a procedure with the given title. DEFINE is a powerful Logo command because it allows a computer program to create a Logo procedure. With DEFINE, programs can write programs! In many versions of Logo, DEFINE is a primitive. LogoWriter, however, does not have DEFINE as a primitive, so we must substitute our own procedure.

Possible Extensions and Expansions

- Increase the list of synonyms for each command.
- TALK.AND.DRAW functions in black and white. A little color could be added by incorporating SETPC and SETBG.
- The error handling can stand to be improved. If there is no command, the program will say that it doesn't understand. However, if a command is identified but there is no argument, the program will crash. Perhaps the program could ask for the argument if none were supplied or understood.
- Modify the program to handle the REPEAT command.
- Recall the example where the user requested that the program "draw a big

square." Fix the program so that it can handle multiword procedure titles. One possible approach would be to have the computer ask what to call the procedure; another might be to concatenate the words so that they become only one word. What other approaches can you think of?

- The program will not necessarily function as desired if the user uses punctuation. If he were to enter "Go ahead 30." the 30 would not be recognized as a number since the word would be seen as thirty period. Modify the program to deal with commas and periods.
- Now, the program "forgets" procedures from one run to the next. SETUP reinitializes :PROCEDURES to the empty list each time the program is run. One solution is not to reinitialize :PROCEDURES; but what happens if you shut off the computer and reload the program? How could you "remember" procedures in that case?
- Besides LEFT and RIGHT, the turtle could be guided by headings or positions on an X-Y coordinate system.
- Program TALK.AND.DRAW so that it recognizes numbers written as words. As the program now stands, 30 would not be identified as the input in the sentence "Go thirty units."

TALK.AND.DRAW
THE PROGRAM
LogoWriter Version

```
TO  TALK.AND.DRAW
SETUP
TD.LOOP
END

TO  SETUP
MAKE  "DEFINING?  "FALSE
MAKE  "FD.LIST  [FORWARD  AHEAD  ADVANCE  FD
  GO  MOVE]
MAKE  "BK.LIST  [BACK  BACKUP  BACKWARDS
  RETREAT  BK]
MAKE  "END.LIST  [END  QUIT  GOODBYE  STOP
  TERMINATE]
MAKE  "CG.LIST  [CLEAR  CLEAN  CG]
MAKE  "RT.LIST  [RIGHT  RT  CLOCKWISE]
MAKE  "LT.LIST  [LEFT  LT]
MAKE  "AND.LIST  [AND  THEN]
MAKE  "ARTICLES  [A  AN  THE]
MAKE  "PROCEDURES  []
MAKE  "COMMAND  [CG]
CT
CG
CC
END

TO  TD.LOOP
IF  :COMMAND  =  [QUIT]  [STOP]
INIT
PARSE.LIST  FIRST  :ENGLISH  :ENGLISH
ACT
TD.LOOP
END


TO  INIT
MAKE  "COMMAND  "
MAKE  "INPUT  []
TYPE  ["=>]
MAKE  "ENGLISH  READLISTCC
END
```

```
TO PARSE.LIST :ATOM :INPUT.LIST
IF :INPUT.LIST = [] [STOP]
IF MEMBER? :ATOM :PROCEDURES [MAKE
  "COMMAND :ATOM MAKE "INPUT []]
IF :ATOM = "DRAW [PROCEDURE.CALL
  :INPUT.LIST
  MAKE "INPUT.LIST [END] STOP]
IF MEMBER? :ATOM :END.LIST [MAKE "COMMAND
  "KWIT ]
IF MEMBER? :ATOM :AND.LIST [ACT MAKE
  "COMMAND " MAKE "INPUT []]
IF NUMBER? :ATOM [MAKE "INPUT :ATOM ]
IF MEMBER? :ATOM :FD.LIST [MAKE "COMMAND
  "FORWARD ]
IF MEMBER? :ATOM :BK.LIST [MAKE "COMMAND
  "BACK ]
IF MEMBER? :ATOM :CG.LIST [MAKE "COMMAND
  "CG ]
IF MEMBER? :ATOM :RT.LIST [MAKE "COMMAND
  "RIGHT ]
IF MEMBER? :ATOM :LT.LIST [MAKE "COMMAND
  "LEFT ]
IF (COUNT :INPUT.LIST) > 1 [PARSE.LIST
  FIRST BUTFIRST :INPUT.LIST BUTFIRST
  :INPUT.LIST]
END

TO ACT
IF :COMMAND = " [(TYPE [I DON'T
  UNDERSTAND] CHAR 13) STOP]
MAKE "COMMAND SENTENCE :COMMAND :INPUT
IF :DEFINING? [MAKE "ROUTINE SENTENCE
  :ROUTINE :COMMAND
RUN :COMMAND
END

TO PROCEDURE.CALL :INPUT.LIST
IFELSE MEMBER? FIRST BUTFIRST :INPUT.LIST
  :ARTICLES
  [MAKE "PROCEDURE FIRST BUTFIRST
  BUTFIRST :INPUT.LIST]
  [MAKE "PROCEDURE FIRST BUTFIRST
  :INPUT.LIST]
IF MEMBER? :PROCEDURE :PROCEDURES [MAKE
  "COMMAND :PROCEDURE
  MAKE "INPUT [] STOP]
(TYPE [TELL ME HOW TO DRAW A] :PROCEDURE
  CHAR 13)
MAKE "DEFINING? "TRUE
MAKE "DONE "FALSE
MAKE "ROUTINE []
PROC.LOOP
MAKE "PROCEDURES SENTENCE :PROCEDURES
  :PROCEDURE
DEFINE :PROCEDURE :ROUTINE
(TYPE :PROCEDURE [IS NOW DEFINED]
  CHAR 13)
MAKE "DEFINING? "FALSE
END

TO PROC.LOOP
INIT
PARSE.PROC FIRST :ENGLISH :ENGLISH
ACT
IF NOT :DONE [PROC.LOOP]
END

TO PARSE.PROC :ATOM :INPUT.LIST
IF :INPUT.LIST = [] [STOP]
IF MEMBER? :ATOM :END.LIST [MAKE "COMMAND
  "STOP
  MAKE "DONE "TRUE STOP]
IF MEMBER? :ATOM :PROCEDURES [MAKE
  "COMMAND :ATOM MAKE "INPUT []]
IF MEMBER? :ATOM :AND.LIST [ACT MAKE
  "COMMAND " MAKE "INPUT []]
IF NUMBER? :ATOM [MAKE "INPUT :ATOM]
IF MEMBER? :ATOM :FD.LIST [MAKE "COMMAND
  "FORWARD ]
IF MEMBER? :ATOM :BK.LIST [MAKE "COMMAND
  "BACK ]
IF MEMBER? :ATOM :CG.LIST [MAKE "COMMAND
  "CG ]
IF MEMBER? :ATOM :RT.LIST [MAKE "COMMAND
  "RIGHT ]
IF MEMBER? :ATOM :LT.LIST [MAKE "COMMAND
  "LEFT ]
IF (COUNT :INPUT.LIST) > 1 [PARSE.PROC
  FIRST BUTFIRST :INPUT.LIST BUTFIRST
  :INPUT.LIST]
END

TO DEFINE :NAME :COMMANDS
IF FRONT? [FLIP]
BOTTOM
PRINT []
PRINT SENTENCE "TO :NAME
PRINT :COMMANDS
PRINT [END]
FLIP
END

TO KWIT
CC
(TYPE [GOODBYE] CHAR 13 CHAR 13)
END
```

TALK.AND.DRAW
THE PROGRAM
Terrapin Logo PLUS Version

```
TO  TALK.AND.DRAW
 SETUP
 TD.LOOP
END

TO  SETUP
 MAKE  "DEFINING?  "FALSE
 MAKE  "FD.LIST  [FORWARD  AHEAD  ADVANCE  FD
  GO  MOVE]
 MAKE  "BK.LIST  [BACK  BACKUP  BACKWARDS
  RETREAT  BK]
 MAKE  "END.LIST  [END  QUIT  GOODBYE  STOP
  TERMINATE]
 MAKE  "CS.LIST  [CLEAR  CLEAN  CS]
 MAKE  "RT.LIST  [RIGHT  RT  CLOCKWISE]
 MAKE  "LT.LIST  [LEFT  LT]
 MAKE  "AND.LIST  [AND  THEN]
 MAKE  "ARTICLES  [A  AN  THE]
 MAKE  "PROCEDURES  []
 DRAW
  CLEARTEXT
END

TO  TD.LOOP
 INIT
 PARSE  FIRST  :ENGLISH  :ENGLISH
 ACT
 TD.LOOP
END

TO  INIT
 MAKE  "COMMAND  "
 MAKE  "INPUT  "
 PRINT1  "=>
 MAKE  "ENGLISH  REQUEST
END

TO  PARSE  :ATOM  :INPUT.LIST
 IF  :INPUT.LIST = []  THEN  STOP
 IF  MEMBER?  :ATOM  :PROCEDURES  THEN  MAKE
  "COMMAND  :ATOM  MAKE  "INPUT  "
 IF  :ATOM = "DRAW  THEN  PROCEDURE.CALL
  :INPUT.LIST
 MAKE  "INPUT.LIST  [END]  STOP
 IF  MEMBER?  :ATOM  :END.LIST  THEN  MAKE
  "COMMAND  "KWIT
 IF  MEMBER?  :ATOM  :AND.LIST  THEN  ACT  MAKE
  "COMMAND  "  MAKE  "INPUT  "
 IF  NUMBER?  :ATOM  THEN  MAKE  "INPUT  :ATOM
 IF  MEMBER?  :ATOM  :FD.LIST  THEN  MAKE
  "COMMAND  "FORWARD
 IF  MEMBER?  :ATOM  :BK.LIST  THEN  MAKE
  "COMMAND  "BACK
 IF  MEMBER?  :ATOM  :CS.LIST  THEN  MAKE
  "COMMAND  "DRAW
 IF  MEMBER?  :ATOM  :RT.LIST  THEN  MAKE
  "COMMAND  "RIGHT
 IF  MEMBER?  :ATOM  :LT.LIST  THEN  MAKE
  "COMMAND  "LEFT
 IF  (COUNT  :INPUT.LIST)  >  1  PARSE  FIRST
  BUTFIRST  :INPUT.LIST  BUTFIRST
  :INPUT.LIST
END

TO  ACT
 IF  :COMMAND = "  THEN  PR  [I  DON'T
  UNDERSTAND]  STOP
 MAKE  "COMMAND  SENTENCE  :COMMAND  :INPUT
 IF  :DEFINING?  THEN  MAKE  "ROUTINE  LPUT
  :COMMAND  :ROUTINE
 RUN  :COMMAND
END

TO  PROCEDURE.CALL  :INPUT.LIST
 IF  MEMBER?  FIRST  BUTFIRST  :INPUT.LIST
  :ARTICLES
 THEN  MAKE  "PROCEDURE  FIRST  BUTFIRST
  BUTFIRST  :INPUT.LIST
 ELSE  MAKE  "PROCEDURE  FIRST  BUTFIRST
  :INPUT.LIST
 IF  MEMBER?  :PROCEDURE  :PROCEDURES  THEN
  MAKE  "COMMAND  :PROCEDURE
 MAKE  "INPUT  "  STOP
 PR  SENTENCE  [TELL  ME  HOW  TO  DRAW  A]
  :PROCEDURE
 MAKE  "DEFINING?  "TRUE
 MAKE  "DONE  "FALSE
 MAKE  "ROUTINE  [[]]
 PROC.LOOP
 MAKE  "PROCEDURES  SENTENCE  :PROCEDURES
  :PROCEDURE
 PR  SENTENCE  :PROCEDURE  [IS  NOW  DEFINED]
 DEFINE  :PROCEDURE  :ROUTINE
 MAKE  "DEFINING?  "FALSE
END

TO  PROC.LOOP
 INIT
 PARSE.PROC  FIRST  :ENGLISH  :ENGLISH
 ACT
 IF  NOT  :DONE  THEN  PROC.LOOP
END
```

```
TO  PARSE.PROC  :ATOM  :INPUT.LIST
  IF  :INPUT.LIST = []  THEN  STOP
  IF  MEMBER?  :ATOM  :END.LIST  THEN  MAKE
   "COMMAND  "STOP
  MAKE  "DONE  "TRUE  STOP
  IF  MEMBER?  :ATOM  :PROCEDURES  THEN  MAKE
   "COMMAND  :ATOM  MAKE  "INPUT  "
  IF  MEMBER?  :ATOM  :AND.LIST  THEN  ACT  MAKE
   "COMMAND  "  MAKE  "INPUT  "
  IF  NUMBER?  :ATOM  THEN  MAKE  "INPUT  :ATOM
  IF  MEMBER?  :ATOM  :FD.LIST  THEN  MAKE
   "COMMAND  "FORWARD
  IF  MEMBER?  :ATOM  :BK.LIST  THEN  MAKE
   "COMMAND  "BACK
  IF  MEMBER?  :ATOM  :CS.LIST  THEN  MAKE
   "COMMAND  "DRAW
  IF  MEMBER?  :ATOM  :RT.LIST  THEN  MAKE
   "COMMAND  "RIGHT
  IF  MEMBER?  :ATOM  :LT.LIST  THEN  MAKE
   "COMMAND  "LEFT
  IF  (COUNT  :INPUT.LIST)  >  1  PARSE.PROC
   FIRST  BUTFIRST  :INPUT.LIST  BUTFIRST
   :INPUT.LIST
END

TO  KWIT
  CLEARTEXT
  PR  [GOODBYE]
  PR  []
  TOPLEVEL
END
```

4 DOCTOR— CONVERSATIONALISTS

Here is an actual conversation. One of the participants is a human, the other is a machine. See if you can determine which is the computer (Simons, 1984).

Where are the cookies?
PERHAPS THE COOKIES ARE ON THE SHELF. SHOULD I CHECK?
Yes.
NO. THE COOKIES ARE NOT ON THE SHELF. MAYBE THEY ARE ON THE COUNTER. I CAN'T FIND THE COOKIES.
I ate the cookies.
O.K.
Where are the cookies?
THERE ARE NONE.

The computer program is a conversationalist. It is familiar with the spelling and definitions of words and knows how to place those words grammatically within a sentence. Such programs are another type of natural language processor. They can take a sentence in English, analyze it for meaning, and then respond to it in a way that reflects understanding. Have you guessed which "person" in the conversation was the computer? If you guessed that it was the first speaker, you were wrong!

This conversationalist program is part of the programming of a robot named LUIGI. LUIGI was designed to help out around the house. He can spot foods and utensils, clean, make sandwiches, and answer questions about the things around him.

Some would say that LUIGI's program has passed one of the first tests designed to check for artificial intelligence. In 1950, Alan Turing, already established as one of the greatest computer scientists, wrote the article "Computer Machinery and Intelligence" (Mishkoff, 1988). In it, he supports his belief that machine intelligence is possible, and he outlines a test to see if a computer has attained human intelligence.

The Turing Test involves three different rooms. Each room is connected to the others with a computer terminal. In one room is an examiner and in the other two are the subjects: one a human, the other a computer. The examiner can converse with the other two only via the computer. Any questions or tasks can be asked of the subjects. If the examiner is uncertain of which is the human after a lengthy interaction, then the machine is considered to have human intelligence.

The test, however, is flawed. Some critics maintain that the test is not rigid enough. A short interaction with a subject can leave the examiner undecided as to whether the subject is human or machine. The conversation with LUIGI could be only two lines away from the robot making some silly mistake that would alert the reader to its true identity. Also, if the examiner were not thorough enough, not asking the subject to answer enough questions or to respond to enough tasks, then an informed decision could not be reached.

Other critics say that the behavior of a program is not the issue. They maintain that although the examiner can ask the subject to write a poem or answer a history question, the test tells us nothing of the process the subject goes through to answer the question. A program could be designed so that the computer simply looks up an appropriate response to any possible inquiry from the examiner. The computer for this program would, however, require an enormous memory. Be-

sides, the critics say, if the program does not process like a human then it does not possess human intelligence. Many programs that might seem to pass the Turing Test but don't use human-like processes simply use tricks to give the appearance of understanding.

One such program is ELIZA. Written by Dr. Joseph Weizenbaum, ELIZA will converse with someone on any one of several topics. One part of the program emulates a Rogerian psychologist. The computer plays the doctor, and the human plays the patient. The Rogerian approach says that the solution to the patient's problem lies within the patient and that the patient will come to the solution through talking over the difficulties. Weizenbaum, then of MIT, wrote ELIZA as a parody of this particular brand of psychology (Hartnell, 1985).

The program had no real understanding of the conversation going on. It used a technique called key-word analysis. It looked through the user's typed message for certain key words and then manipulated the sentence to make it sound like the computer was responding intelligently. The tricks were really quite simple. If the word "sister" appeared in the user's text, the computer might respond with "Tell me more about your family." At other times, the computer would change the sentence around by substituting "you" for "me" and "yours" for "my." If the user's sentence was "I think they don't like me," the computer would swap "What makes you think" for "I think" and produce "What makes you think they don't like you?"

Many programs are written using algorithms. Algorithms are "recipes" and, when well designed, always produce the desired result. For example, if you wanted to find a name in a list, one algorithm is to start at the beginning of the list and look at each name until you find the one you wanted. This recipe will always find your name, as long as it is in the list.

Instead of an algorithm, the system of looking through sentences for key words is an example of a style of programming that is frequently used in artificial intelligence: heuristics. Heuristics can be thought of as "rules of thumb." They are home-made recipes that don't always produce a desired response. (The term "rule of thumb" came from the unwritten rule that teachers shouldn't cut a switch for disciplining students that was any larger around than their thumb!) The rule "I before E except after C" is a heuristic. Usually this rule of thumb will help a writer come up with a correct spelling, but the rule falls short on words like "neighbor" and "weigh."

Rules of thumb are used for AI programming for several reasons. Recipes guaranteeing success are often large and slow. Although computer memory and speed have been issues for a long time, the cost of each is dropping at the same time that capacity is increasing. As affordable memory and speed grow, so does the power of the tools the computer owner can purchase. But memory and speed are not the only reason for using heuristics instead of algorithms. Many times scientists just don't know the right recipe for success. For example, an algorithm designed to process natural language would certainly be enormous because of the large vocabulary and complex grammar of human languages. In fact, the algorithm cannot be written because linguists, psychologists, and computer scientists don't know how people really process language.

How does the heuristic of key-word analysis and fixed responses fit into the study of intelligence? Many researchers have serious reservations about how programs that give fixed replies to recognized patterns do fit in. Some might say that this behavior does not mimic human behavior. Perhaps, however, simple recognition of patterns and a fixed range of preset re-

sponses really is a part of human intelligence. We certainly want people to react to, as opposed to think about, some questions. When we ask, "What is 12 plus 5?" we want a student to reply instantly with "17." We think it a sign of less ability if the student has to stop and think about the answer.

Mathematics is a specialized area, though, and we may not want a person simply to react during a conversation. Nevertheless, this is how some people behave. Many people "react" to simple greetings ("Hi." "How are you?") but we may label a person who regularly reacts unthinkingly during a conversation as not paying attention or possibly not being very intelligent. We must recognize, however, that this behavior is, at least in some small way, a part of human intelligence.

Furthermore, recall that not all artificial intelligence researchers feel that the purpose of studying AI is to model human behavior and thought. Many believe that creating useful programming techniques is sufficient reason for the pursuit. In this light, programs like ELIZA offer some insight into creating friendly interfaces between humans and machines.

Dr. Weizenbaum's program was a friendly interface, and he was quickly surprised by the reaction of those around him. His student assistants and his secretary used the program frequently. He was fascinated and wanted to know more about their interactions, but they were secretive and didn't want him to know what personal problems they were discussing with ELIZA. The secretary even once responded that she wanted a private interview with ELIZA! They found their conversations with ELIZA refreshing and treated the interactions as if the program were a long-time confidant. Dr. Weizenbaum was further perplexed that some people felt that ELIZA really understood what it was discussing (Hartnell, 1985; Simons, 1984).

Weizenbaum's criticisms run deep. Weizenbaum meant his program to be a simple parody. He was shocked by those around him who acted as if ELIZA were real (Hartnell, 1985; Peat, 1988; Chabris, 1987). Weizenbaum became a significant critic of computers in society. He does not feel that computers should be used in some parts of society because people are so easily fooled by the machine, putting blind trust in its decisions and apparently being unable to objectively evaluate the machine's actions. He and many others, including John Kemeny, the creator of BASIC, believe that there is real danger in putting unquestioned power so easily in the hands of machines.

Weizenbaum became even more concerned when someone suggested that his program could be used to treat patients. Kenneth Colby, a professor of Psychiatry at UCLA, felt that an expanded version of Weizenbaum's Doctor could be made available to patients in mental hospitals (Hartnell, 1985; Krutch, 1986; Chabris, 1987). Weizenbaum became outraged. ELIZA had no real understanding, and he felt patients need interaction with trained, insightful experts. Weizenbaum found Colby's suggestion downright dangerous.

Colby later created another conversational program called PARRY. It behaved as a paranoid schizophrenic. Colby had psychiatrists compare a PARRY transcript with a conversation with a real patient. He had the doctors try to determine which conversation was with the computer and which was with the patient. Only about fifty percent chose correctly. That is about the same chance as guessing and indicates that PARRY is a pretty good simulation of a paranoid patient. Below is a sample conversation (Krutch, 1986). The human is in mixed case letters and the computer is in upper case.

Why are you in the hospital?
I SHOULDN'T BE HERE
Who brought you here?
THE POLICE.
What trouble did you have with the police?
COPS DON'T DO THEIR JOB.
What do they do instead?
COPS ARREST THE WRONG PEOPLE.
They arrest the right people sometimes.
THE LAW SHOULD GET THOSE ITALIAN CROOKS.
Are you Italian?
MY BACKGROUND IS BRITISH BUT I WAS BORN IN THIS COUNTRY.
Do you know any Italian crooks?
I TRY TO AVOID THE UNDERWORLD.
Are you afraid of the underworld?
THE MAFIA ARE THE WORST GANGSTERS.
Are you connected with the mafia?
YOU KNOW, THEY KNOW ME.
How do they know you?
THEY SPY ON ME.
Why do they?
THEY WANT TO GET FALSE EVIDENCE TO FRAME ME.
What do they have against you?
THEY ARE OUT TO GET ME.
What for?
I HAVE INCRIMINATING EVIDENCE AGAINST THE MAFIA.

PARRY, like many computerized conversationalists, sometimes makes inappropriate responses. It can get away with it, however, since paranoids occasionally do make inappropriate statements. Still, both ELIZA and PARRY simply reply in a preset fashion to recognized patterns of letters. This supports the criticism that many have about the accuracy of the Turing Test of machine intelligence.

It was only a matter of time before someone thought to put PARRY and ELIZA together. Here is one conversation (Simons, 1984):

PARRY: IN MY OPINION YOU HAVE A ONE-TRACK MIND.
ELIZA: WHAT MAKES YOU THINK I HAVE A ONE-TRACK MIND?
PARRY: I USED TO GO TO THE TRACKS EVERY WEEK.
ELIZA: WHAT DOES THAT SUGGEST TO YOU?
PARRY: I AVOID RACETRACKS NOW BECAUSE THE RACES ARE FIXED.
ELIZA: IS THAT THE REAL REASON?
PARRY: I'D RATHER NOT DISCUSS IT.
ELIZA: WHAT ELSE WOULD YOU LIKE TO DISCUSS?
PARRY: THAT'S SOMETHING I DON'T WANT TO GO INTO.

The conversation quickly digresses and becomes uninteresting. Each conversationalist seems to require a person's intelligence to keep the conversation going.

The Program

Whatever your perspective on where ELIZA fits into the study of AI, it is still an important program to examine. Several books give listings for versions of ELIZA written in BASIC (Hartnell, 1985; Krutch, 1986) and a few in Logo (Abelson, 1982; Harvey, 1987). Our version of ELIZA is called DOCTOR. Let's look at a sample dialogue with DOCTOR. Once again, the computer response is in all capital letters and the human response is in mixed case.

THE DOCTOR IS IN

WHAT SEEMS TO BE THE TROUBLE
I think my teacher is strange

WHAT MAKES YOU THINK YOUR TEACHER IS STRANGE
The other day she was whistling in class

YOU SAY THE OTHER DAY SHE WAS WHISTLING IN CLASS
She said that she was glad it was Spring

PLEASE GO ON
I believe she is daffy

DO YOU DOUBT SHE IS DAFFY
No

THAT'S VERY INTERESTING

DOCTOR can really leave you with the impression of understanding. But remember, this program functions in a way that many feel is far from intelligent. It uses the trick of looking for key words, then responding with some prefabricated response. This can sometimes backfire. The conversation above continues:

THAT'S VERY INTERESTING
I want her to stop acting silly

WHAT WOULD YOU GAIN IF YOU WERE HER TO STOP ACTING SILLY

The program took my input and responded to it but the response is nonsense. Watch as the program quickly returns to apparent understanding but then makes two mistakes:

WHAT WOULD YOU GAIN IF YOU WERE HER TO STOP ACTING SILLY
Maybe I should just switch classes

PLEASE GO ON
I am the most depressed I can remember

DID YOU COME TO ME BECAUSE YOU ARE THE MOST DEPRESSED YOU CAN REMEMBER
Yes

YOU SEEM VERY CERTAIN ABOUT IT
Wouldn't you

YOU SEEM TO THINK WOULDN'T YOU
I think you are a jerk!

WHAT MAKES YOU THINK YOU ARE A JERK!

At least the DOCTOR's mistakes can occasionally be amusing!

As stated earlier, DOCTOR works using a process called key-word analysis. It takes the user's input, and after some fudging, places it in a response. The response is chosen based on certain words found in the user's input. The whole process is managed from two procedures: DOCTOR.LOOP and PARSE.AND.ACT.

```
TO DOCTOR.LOOP
MAKE "INPUT READLIST
PRINT []
IF :INPUT = [GOODBYE] [PRINT [COME FOR
  ANOTHER VISIT SOON] STOP]
PARSE.AND.ACT
DOCTOR.LOOP
END
```

```
TO  PARSE.AND.ACT
MAKE  "RESPONSE.FOUND  "FALSE
IF  (FRONT  2  :INPUT)  =  [I  AM]
   [BE.RESPONSE]
IF  (FRONT  2  :INPUT)  =  [I  BELIEVE]
   [BELIEVE.RESPONSE]
IF  (FRONT  2  :INPUT)  =  [I  THINK]
   [THINK.RESPONSE]
IF  (FRONT  2  :INPUT)  =  [I  FEEL]
   [FEEL.RESPONSE]
IF  (FRONT  2  :INPUT)  =  [I  WANT]
   [WANT.RESPONSE]
IF  (FRONT  1  :INPUT)  =  [YES]
   [YES.RESPONSE]
IF  (FRONT  1  :INPUT)  =  [MAYBE]
   [MAYBE.RESPONSE]
IF  NOT  :RESPONSE.FOUND  [CHOOSE.RESPONSE]
END
```

FRONT is similar to the primitive FIRST, but instead of taking only the very first item in a list, FRONT allows the programmer to specify how many items to take. FRONT is used here to locate the key words. You can see that the phrases the computer is looking for are "I am..." "I believe..." "I think..." "I feel..." "I want..." "Yes..." and "Maybe..." Each key phrase has a procedure that creates a response.

Typically, the key phrase is first stripped away. Next, the remaining part of the sentence is changed from first person to second person and the computer's own leading phrase is stuck on the front. Each key phrase triggers a different leading phrase from the computer.

KEY PHRASE	COMPUTER'S LEADING PHRASE
I am...	Did you come to me because you are...
I believe...	Do you doubt...
I think...	What makes you think...
I feel...	Do you often feel...
I want...	What would you gain if you were...

As an example, if the user were to type "I believe someone is out to get me" the program would just remove the phrase "I believe," change the word "me" to "you" and tag on the phrase "Do you doubt." The response would be "Do you doubt someone is out to get you."

Changing first person to second person, for example, "me" to "you," involves several procedures. CHANGE.PERSON calls another general-purpose procedure called SUBST. SUBST takes two inputs. In general, the first input is a list and the second is a table (list of lists) of pairs of items. In our example, the first input is the remaining part of the user's sentence and the second input is the variable :PRONOUNS. :PRONOUNS is the list of pronouns and what they are to be swapped with; "I" is to be exchanged with "you," and "my" with "your," etc. SUBST.ITEM completes the exchange.

```
TO  CHANGE.PERSON  :PHRASE
OUTPUT  SUBST  :PHRASE  :PRONOUNS
END

TO  SUBST  :LIST  :TABLE
IF  :LIST  =  []  [OUTPUT  []]
OUTPUT  FPUT  SUBST.ITEM  FIRST  :LIST  :TABLE
   SUBST  BUTFIRST  :LIST  :TABLE
END

TO  SUBST.ITEM  :ITEM  :TABLE
IF  :TABLE  =  []  [OUTPUT  :ITEM]
IF  :ITEM  =  FIRST  FIRST  :TABLE  [OUTPUT
   LAST  FIRST  :TABLE]
OUTPUT  SUBST.ITEM  :ITEM  BUTFIRST  :TABLE
END
```

When the user's sentence begins with either "yes" or "maybe," a different type of response is generated. Based on a random number, the computer sometimes replies with "You seem very certain about it" if the keyword is "yes," or "You don't seem quite certain" if the keyword is "maybe." Other

times, the computer generates a hedge, such as "Please go on" or "Please elaborate on that."

These short responses, used to urge the user to continue, are selected by HEDGE. It uses a procedure called PICK, which randomly chooses a phrase from the list called :HEDGES. This includes short sentences such as "Please go on," "That's very Interesting," or "Please elaborate on that."

```
TO HEDGE
PRINT PICK :HEDGES
END

TO PICK :OBJECT
OUTPUT ITEM (1 + RANDOM COUNT :OBJECT)
  :OBJECT
END

TO CHOOSE.RESPONSE
IFELSE (RANDOM 4) = 0 [HEDGE] [RESPOND
  :INPUT]
END

TO RESPOND :INPUT
PRINT SENTENCE PICK :QUALIFY
  CHANGE.PERSON :INPUT
END
```

If an applicable response is not found, CHOOSE.RESPONSE is called. Sometimes it produces another hedge, but most of the time it calls the procedure RESPOND. This routine changes the person of the user's sentence, then tags onto the front a randomly selected leading phrase. These leading phrases are stored in :QUALIFY and include the phrases "You seem to think...," "You feel that...," and "You say...."

Remember that although the program can often seem as if it understands, there are still many times when the computer's responses do not seem appropriate. The more contingencies you plan for and build into the program, the more intelligent the program will appear to be.

Possible Extensions and Expansions

- Have the computer deal with punctuation. Have the punctuation removed from the user's input and add it to the computer's responses.
- Sometimes a user might start a sentence with a leading phrase, such as "sometimes," "well," or "you see." This can prevent the program from noticing the key words that would otherwise be at the beginning of the user's sentence. One solution is to remove these leading phrases before the sentence is sent to PARSE.AND.ACT.
- Increase the number of responses the computer might generate for each of the types of responses the program already understands. You could store each set of leading phrases in a variable and use PICKRANDOM to select one.
- Have DOCTOR look for other key words and respond accordingly. Some versions of ELIZA are triggered by the word "computer" and others have long sets of responses included for use when the user mentions a member of the family.
- Have the computer remember pieces of the conversation and bring them up again. Some versions of ELIZA just bring things up at random to change the conversation.
- You have seen two examples of programs that use key-word analysis to respond to the user. Create your own. Perhaps it could be a whiny little brother or a police officer or a school teacher.

DOCTOR
THE PROGRAM
LogoWriter Version

```
TO DOCTOR
INITIALIZE
PRINT [THE DOCTOR IS IN]
PRINT []
PRINT [WHAT SEEMS TO BE THE TROUBLE]
DOCTOR.LOOP
END

TO INITIALIZE
MAKE "HEDGES [[PLEASE GO ON] [THAT'S VERY
  INTERESTING]
  [PLEASE ELABORATE ON THAT]]
MAKE "PRONOUNS [[I YOU] [ME YOU]
  [MY YOUR] [AM ARE]]
MAKE "QUALIFY [[YOU SEEM TO THINK] [YOU
  FEEL THAT] [YOU SAY]]
CT
CG
HT
END

TO DOCTOR.LOOP
MAKE "INPUT READLIST
PRINT []
IF :INPUT = [GOODBYE] [PRINT [COME FOR
  ANOTHER VISIT SOON] STOP]
PARSE.AND.ACT
DOCTOR.LOOP
END

TO PARSE.AND.ACT
MAKE "RESPONSE.FOUND "FALSE
IF (FRONT 2 :INPUT) = [I AM]
  [BE.RESPONSE]
IF (FRONT 2 :INPUT) = [I BELIEVE]
  [BELIEVE.RESPONSE]
IF (FRONT 2 :INPUT) = [I THINK]
  [THINK.RESPONSE]
IF (FRONT 2 :INPUT) = [I FEEL]
  [FEEL.RESPONSE]
IF (FRONT 2 :INPUT) = [I WANT]
  [WANT.RESPONSE]
IF (FRONT 1 :INPUT) = [YES]
  [YES.RESPONSE]
IF (FRONT 1 :INPUT) = [MAYBE]
  [MAYBE.RESPONSE]
IF NOT :RESPONSE.FOUND [CHOOSE.RESPONSE]
END

TO FRONT :OBJECT :LIST
IF :OBJECT < 1 [OUTPUT [] STOP]
IF (COUNT :LIST) < :OBJECT [OUTPUT :LIST
  STOP]
OUTPUT SENTENCE FIRST :LIST
  FRONT :OBJECT - 1 BUTFIRST :LIST
END

TO CHANGE.PERSON :PHRASE
OUTPUT SUBST :PHRASE :PRONOUNS
END

TO SUBST :LIST :TABLE
IF :LIST = [] [OUTPUT []]
OUTPUT FPUT SUBST.ITEM FIRST :LIST :TABLE
  SUBST BUTFIRST :LIST :TABLE
END

TO SUBST.ITEM :ITEM :TABLE
IF :TABLE = [] [OUTPUT :ITEM]
IF :ITEM = FIRST FIRST :TABLE [OUTPUT
  LAST FIRST :TABLE]
OUTPUT SUBST.ITEM :ITEM BUTFIRST :TABLE
END

TO HEDGE
PRINT PICK :HEDGES
END

TO PICK :OBJECT
OUTPUT ITEM (1 + RANDOM COUNT :OBJECT)
  :OBJECT
END

TO RESPOND :INPUT
PRINT SENTENCE PICK :QUALIFY
  CHANGE.PERSON :INPUT
END

TO BE.RESPONSE
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
PRINT SENTENCE [DID YOU COME TO ME
  BECAUSE YOU ARE] CHANGE.PERSON :INPUT
MAKE "RESPONSE.FOUND "TRUE
END

TO BELIEVE.RESPONSE
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
PRINT SENTENCE [DO YOU DOUBT]
  CHANGE.PERSON :INPUT
MAKE "RESPONSE.FOUND "TRUE
END
```

```
TO THINK.RESPONSE
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
PRINT SENTENCE [WHAT MAKES YOU THINK]
  CHANGE.PERSON :INPUT
MAKE "RESPONSE.FOUND "TRUE
END

TO FEEL.RESPONSE
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
PRINT SENTENCE [DO YOU OFTEN FEEL]
  CHANGE.PERSON :INPUT
MAKE "RESPONSE.FOUND "TRUE
END

TO WANT.RESPONSE
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
PRINT SENTENCE [WHAT WOULD YOU GAIN IF
  YOU WERE] CHANGE.PERSON :INPUT
MAKE "RESPONSE.FOUND "TRUE
END

TO MAYBE.RESPONSE
IFELSE (RANDOM 2) = 0 [PRINT [YOU DON'T
  SEEM QUITE CERTAIN]] [HEDGE]
MAKE "RESPONSE.FOUND "TRUE
END

TO CHOOSE.RESPONSE
IFELSE (RANDOM 4) = 0 [HEDGE] [RESPOND
  :INPUT]
END

TO YES.RESPONSE
IFELSE (RANDOM 2) = 0 [PRINT [YOU SEEM
  VERY CERTAIN ABOUT IT]] [HEDGE]
MAKE "RESPONSE.FOUND "TRUE
END
```

DOCTOR
THE PROGRAM
Terrapin Logo PLUS Version

```
TO DOCTOR
  INITIALIZE
 PRINT [THE DOCTOR IS IN]
 PRINT []
 PRINT [WHAT SEEMS TO BE THE TROUBLE]
  DOCTOR.LOOP
END

TO INITIALIZE
 MAKE "HEDGES [[PLEASE GO ON] [THAT'S
  VERY INTERESTING]
 [PLEASE ELABORATE ON THAT]]
 MAKE "PRONOUNS [[I YOU] [ME YOU]
  [MY YOUR] [AM ARE]]
 MAKE "QUALIFY [[YOU SEEM TO THINK] [YOU
  FEEL THAT] [YOU SAY]]
  CLEARTEXT
END

TO DOCTOR.LOOP
 MAKE "INPUT REQUEST
 PRINT []
 IF :INPUT = [GOODBYE] PRINT [COME FOR
  ANOTHER VISIT SOON] STOP
 PARSE.AND.ACT
 DOCTOR.LOOP
END

TO PARSE.AND.ACT
  MAKE "RESPONSE.FOUND "FALSE
 IF FRONT 2 :INPUT = [I AM] THEN
  BE.RESPONSE
 IF FRONT 2 :INPUT = [I BELIEVE] THEN
  BELIEVE.RESPONSE
 IF FRONT 2 :INPUT = [I THINK] THEN
  THINK.RESPONSE
 IF FRONT 2 :INPUT = [I FEEL] THEN
  FEEL.RESPONSE
 IF FRONT 2 :INPUT = [I WANT] THEN
  WANT.RESPONSE
 IF FRONT 1 :INPUT = [YES] THEN
  YES.RESPONSE
 IF FRONT 1 :INPUT = [MAYBE] THEN
  MAYBE.RESPONSE
 IF NOT :RESPONSE.FOUND THEN
  CHOOSE.RESPONSE
END
```

```
TO FRONT :OBJECT :LIST
 IF :OBJECT < 1 THEN OUTPUT [] STOP
 IF COUNT :LIST < :OBJECT THEN OUTPUT
  :LIST STOP
 OUTPUT SENTENCE FIRST :LIST FRONT
  :OBJECT - 1 BUTFIRST :LIST
END

TO CHANGE.PERSON :PHRASE
 OUTPUT SUBST :PHRASE :PRONOUNS
END

TO SUBST :LIST :TABLE
 IF :LIST = [] OUTPUT []
 OUTPUT FPUT SUBST.ITEM FIRST :LIST
  :TABLE SUBST BUTFIRST :LIST :TABLE
END

TO SUBST.ITEM :ITEM :TABLE
 IF :TABLE = [] OUTPUT :ITEM
 IF :ITEM = FIRST FIRST :TABLE OUTPUT
  LAST FIRST :TABLE
 OUTPUT SUBST.ITEM :ITEM BUTFIRST :TABLE
END

TO HEDGE
 PRINT PICK :HEDGES
END

TO PICK :OBJECT
 OUTPUT ITEM (1 + RANDOM COUNT :OBJECT)
  :OBJECT
END

TO RESPOND :INPUT
 PRINT SENTENCE PICK :QUALIFY
  CHANGE.PERSON :INPUT
END

TO BE.RESPONSE
 MAKE "INPUT BUTFIRST BUTFIRST :INPUT
 PRINT SENTENCE [DID YOU COME TO ME
  BECAUSE YOU ARE]
 CHANGE.PERSON :INPUT
 MAKE "RESPONSE.FOUND "TRUE
END

TO BELIEVE.RESPONSE
 MAKE "INPUT BUTFIRST BUTFIRST :INPUT
 PRINT SENTENCE [DO YOU DOUBT]
  CHANGE.PERSON :INPUT
 MAKE "RESPONSE.FOUND "TRUE
END

TO THINK.RESPONSE
 MAKE "INPUT BUTFIRST BUTFIRST :INPUT
 PRINT SENTENCE [WHAT MAKES YOU THINK]
  CHANGE.PERSON :INPUT
 MAKE "RESPONSE.FOUND "TRUE
END

TO FEEL.RESPONSE
 MAKE "INPUT BUTFIRST BUTFIRST :INPUT
 PRINT SENTENCE [DO YOU OFTEN FEEL]
  CHANGE.PERSON :INPUT
 MAKE "RESPONSE.FOUND "TRUE
END

TO WANT.RESPONSE
 MAKE "INPUT BUTFIRST BUTFIRST :INPUT
 PRINT SENTENCE [WHAT WOULD YOU GAIN IF
  YOU WERE] CHANGE.PERSON :INPUT
 MAKE "RESPONSE.FOUND "TRUE
END

TO MAYBE.RESPONSE
 IF (RANDOM 2) = 0 PRINT [YOU DON'T SEEM
  QUITE CERTAIN] ELSE HEDGE
 MAKE "RESPONSE.FOUND "TRUE
END

TO CHOOSE.RESPONSE
 IF (RANDOM 4) = 0 HEDGE ELSE RESPOND
  :INPUT
END

TO YES.RESPONSE
 IF (RANDOM 2) = 0 PRINT [YOU SEEM VERY
  CERTAIN ABOUT IT] ELSE HEDGE
 MAKE "RESPONSE.FOUND "TRUE
END
```

5 BLOCKS— MACHINE INTELLIGENCE

As with other sciences, there is a pure and an applied side to artificial intelligence programming and research. Pure AI programmers study AI programs to learn about the mind, and study the mind to develop programs. They are interested in producing programs that behave intelligently. The applied AI researchers look for ways to use artificial intelligence techniques in commercial and application programs. As with other sciences, an aspect of AI that has commercial application is more likely to receive funding than one that is of little or no practical use.

Natural language processing continues to be an artificial intelligence area that is commercially profitable. Many databases, for example, are capable of searching for information based on a request in English. Suppose a company database contains information on living and sales conditions in all the sales regions in the country. You have the opportunity of selecting which sales region you wish to be next assigned. Many databases (*Q&A*, for example) will now allow you to enter your query as a sentence. You could type: WHERE IN THE MIDWEST IS THE SALES REGION OF HIGHEST PROFIT WITH THE SCHOOLS RATED AT LEAST GOOD AND COST OF LIVING IS LESS THAN 5% ABOVE AVERAGE. The program would then analyze your sentence and respond.

Another area of natural language processing, which experts had hoped would be profitable, is Machine Translation Systems (MT). It was long believed that machines would replace human translators. The complexities of human language, however, have severely limited the success of MT experts. Large vocabularies and the complex patterns allowed by human language make it almost impossible to accurately translate all passages. Experts have had great success when the output produced by the machine is used simply as a draft. Pretranslation editing can also improve the results.

SYSTRAN, developed by Xerox, is used to translate documents from French into English. *Cult* is a MT system used in Hong Kong to translate mathematics journals. Both systems require that papers be pre-edited and that the translations be post-edited to get the best results. There are about 20 Machine Translation systems used world-wide.

These translator systems look at the words in the original language and then substitute the appropriate word from the target language. Substitution is a trick similar to the tricks that programs like Doctor use. These tricks give the impression that the computer is intelligent. It raises some interesting questions about what is true intelligence. The answer seems to lie in how the program responds.

Klir and Valach suggest that understanding natural language has taken place if, when responding to a message, the response can be interpreted as showing that understanding has taken place. Geoff Simons thinks the computer has the ability to understand language if the computer can answer questions or make intelligent connections between the message and the data in its program, called the knowledge base (Hartnell, 1985). Another researcher of natural language processing has a similar answer. Wendy Lehnert writes (Krutch, 1986):

> When people understand stories, they can demonstrate their understanding by answering questions about the story. Because questions can be devised to query any aspect of text comprehension, the ability to answer questions is the strongest possible demonstration of understanding.... If a computer is said to understand a story, we must demand of the computer the same demonstrations of understanding that we require of people. Until such demands are met, we have no way of evaluating text-understanding programs. Any computer programmer can write a program that inputs text. If the programmer assures us that the program "understands" text, it is a bit like being reassured by a used car dealer about a suspiciously low speedometer reading. Only when we can ask a program to answer questions about what it reads will we be in a position to assess that program's comprehension.

There still exist, however, the problems of large vocabularies and multiple meanings. One way of dealing with these problems is to limit the domain of the program. This is to say that by creating a microworld in which the program operates, the program can easily be designed to respond intelligently.

One such program is still considered to be a true landmark in artificial intelligence. The world of SHRDLU, written by Terry Winograd in 1972, consists of blocks and pyramids of various sizes and colors. They can be moved around, or the user can ask the computer questions about the blocks. The program has an extensive vocabulary and is really quite impressive. Below are several parts of a sample conversation with SHRDLU (Hartnell, 1985). Once again, the computer response is in all caps. The questioning takes place after the user moved the blocks around for a while. Watch how well the computer seems to understand the human.

Pick up the big red block.
>OK
Grasp the pyramid.
>I DON'T UNDERSTAND WHICH PYRAMID YOU MEAN
Find the block which is taller than the one you are holding and put it into the box
>BY "IT", I ASSUME YOU MEAN THE BLOCK WHICH IS TALLER THAN THE ONE I HAVE
>OK

(Later, after moving some things around...)

How many things are on top of green cubes?
>I'M NOT SURE WHAT YOU MEAN BY "ON TOP OF" IN THE PHRASE "ON TOP OF GREEN CUBES"
>DO YOU MEAN:
1 - DIRECTLY ON THE SURFACE
2 - ANYWHERE ON TOP OF

(If that's not impressive enough...)

Had you touched any pyramid before you put the green one on the little cube?
>YES, THE GREEN ONE
When did you pick it up?
>WHEN I WAS STACKING UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE
Why?
>TO GET RID OF IT
Why did you do that?
>TO CLEAR OFF THE RED CUBE

Why did you clear off that cube?
>TO PUT IT ON A LARGE GREEN CUBE
Why did you do that?
>TO STACK UP THE RED CUBE, A LARGE RED BLOCK AND A LARGE GREEN CUBE
Why did you do that?
>BECAUSE YOU ASKED ME TO

The Program

Programs that are scaled down versions of SHRDLU appear in several texts (Weston, 1985; Friendly, 1988; Hartnell, 1985). Our program, BLOCKS, is also a miniature version of SHRDLU. In BLOCKS, the microworld consists of blocks of different colors. The computer can respond to messages typed in English (albeit a very limited subset of the language) by the user and make the appropriate response. These blocks live in a two-dimensional world that measures six spaces by five spaces. Let's examine a conversation with BLOCKS.

```
WELCOME TO THE WORLD OF BLOCKS

   ......
   ......
   ......
   ......
   .RWBG.

>Put the blue block on the red one
MOVING
OBJECT FOUND IN COLUMN 4
LEVEL 1
TARGET FOUND IN COLUMN 2
LEVEL 1

   ......
   ......
   ......
   .B....
   .RW.G.

>Put the green on the white block
MOVING
OBJECT FOUND IN COLUMN 5
LEVEL 1
TARGET FOUND IN COLUMN 3
LEVEL 1

    ......
    ......
    ......
    .BG...
    .RW...

>Tell me what you see

STARTING FROM THE LEFT

... A SPACE
IT'S THE BLUE BLOCK
AND BELOW THAT...
IT'S THE RED BLOCK
IT'S THE GREEN BLOCK
AND BELOW THAT...
IT'S THE WHITE BLOCK
... A SPACE
... A SPACE
... A SPACE

    ......
    ......
    ......
    .BG...
    .RW...

>Where is the green block
LOOKING

FOUND IN COLUMN 3 LEVEL 2

NOTHING ABOVE THAT
BELOW THE GREEN ONE IS THE WHITE
TO THE LEFT, THE BLUE ONE

   ......
   ......
   ......
   .BG...
   .RW...
```

(Later, after some things have been moved around a bit)

```
   ......
   ......
   ....G.
   ....R.
   ..W.B.

>Put the white block on the white one
I CAN'T DO THAT

   ......
   ......
   ....G.
   ....R.
   ..W.B.

>Put the white on the green one
MOVING
OBJECT FOUND IN COLUMN 3 LEVEL 1
TARGET FOUND IN COLUMN 5 LEVEL 3

   ......
   ....W.
   ....G.
   ....R.
   ....B.

>Tell me what you see

STARTING FROM THE LEFT

... A SPACE
... A SPACE
... A SPACE
... A SPACE
IT'S THE WHITE BLOCK
AND BELOW THAT...
IT'S THE GREEN BLOCK
AND BELOW THAT...
IT'S THE RED BLOCK
AND BELOW THAT...
IT'S THE BLUE BLOCK
... A SPACE

   ......
   ....W.
   ....G.
   ....R.
   ....B.
```

(Watch what happens now as we ask it to do some rearranging. The computer will have to move away several of the blocks to complete the task. The program does this by selecting a random location to put the block. Nothing is displayed on the screen, but this would make a good addition to the program.)

```
>Put the white block on the red one
MOVING
MOVING THE WHITE BLOCK OUT OF THE WAY
MOVING THE GREEN BLOCK OUT OF THE WAY
OBJECT FOUND IN COLUMN 2 LEVEL 1
TARGET FOUND IN COLUMN 5 LEVEL 2

   ......
   ......
   ....W.
   ....R.
   ...GB.
```

(Let's try one that is even tougher. What if we ask it to put the block on the bottom of the pile on another block? First, let's rebuild the tower.)

```
>Put the green on the white block
MOVING
OBJECT FOUND IN COLUMN 4 LEVEL 1
TARGET FOUND IN COLUMN 5 LEVEL 3

   ......
   ....G.
   ....W.
   ....R.
   ....B.

>Put the blue block on the green one
MOVING
MOVING THE GREEN BLOCK OUT OF THE WAY
MOVING THE WHITE BLOCK OUT OF THE WAY
MOVING THE RED BLOCK OUT OF THE WAY
OBJECT FOUND IN COLUMN 5 LEVEL 1
TARGET FOUND IN COLUMN 3 LEVEL 1

    ......
    ......
    ......
    ..B...
    R.GW..

>Tell me what you see
```

```
STARTING FROM THE LEFT

IT'S THE RED BLOCK
... A SPACE
IT'S THE BLUE BLOCK
AND BELOW THAT...
IT'S THE GREEN BLOCK
IT'S THE WHITE BLOCK
... A SPACE
... A SPACE


    ......
    ......
    ......
    ..B...
    R.GW..

>Where is the green block
LOOKING

FOUND IN COLUMN 3 LEVEL 1

ABOVE THAT, I SEE THE BLUE ONE
NOTHING ABOVE THAT
TO THE RIGHT, THE WHITE BLOCK
```

Notice that this comes close to our definition of machine intelligence. When the user tells the computer to move a piece, the computer responds by moving the correct block. When questioned about where a block is, the computer can describe the location of that piece. BLOCKS does not yet have the wide scope of understanding that SHRDLU has, but the framework is there, ready for expansion.

Let's examine some of the key procedures that make this program work. The secret to BLOCKS working is the 5x6 microworld and the data structure that represents it. Here we will use a list of six lists, each sublist representing one column. The line in the initialization procedure that puts one block in each of the middle four columns looks like this:

```
MAKE "WORLD [[. . . . .] [R . . . .]
  [W . . . .] [B . . . .]
  [G . . . .] [. . . . .]]
```

When :WORLD is printed, the grid looks like this:

```
......
......
......
......
.RWBG.
```

You should be very careful when typing the line in the initialization procedure to make sure there is a space between each color and each period. The program will not work properly without these spaces.

Printing the world is handled by two procedures called PRINTWORLD and PRINTWORLD.LOOP. PRINTWORLD.LOOP prints each column from the top down. In PRINTWORLD.LOOP, the variable LEVEL, which PRINTWORLD initializes to five, is used to determine which level is being printed. The primitive ITEM is used to get the proper row and then again to get the item at the proper level.

```
TO PRINTWORLD
PRINT [] PRINT []
PRINTWORLD.LOOP 5
PRINT [] PRINT []
END

TO PRINTWORLD.LOOP :LEVEL
IF :LEVEL < 1 [STOP]
(PRINT (ITEM :LEVEL ITEM 1 :WORLD) (ITEM
  :LEVEL ITEM 2 :WORLD)
  (ITEM :LEVEL ITEM 3 :WORLD) (ITEM
  :LEVEL ITEM 4 :WORLD)
  (ITEM :LEVEL ITEM 5 :WORLD) (ITEM
  :LEVEL ITEM 6 :WORLD) )
PRINTWORLD.LOOP :LEVEL - 1
END
```

After the program gets the user's sentence with the procedure GETUSERINPUT, PARSE.AND.ACT is called. This procedure examines the user's sentence and decides what action to take. There are three types of actions. The program will either find a specific block and describe its surroundings, describe where all the blocks are, or put one block on top of another.

FRONT takes the specified number of words from the front of input. If the sentence begins with "Where is" the program will find the one block. If the first three words are "Tell me what" then the program tells where all the blocks are. When the first words are "Put the" the computer moves the described block.

```
TO  PARSE.AND.ACT
IF  (FRONT 2 :INPUT) = [WHERE IS]
   [FIND.BLOCK]
IF  (FRONT 3 :INPUT) = [TELL ME WHAT]
   [DESCRIBE.WORLD]
IF  (FRONT 2 :INPUT) = [PUT THE]
   [MOVE.BLOCKS]
IF  (FRONT 1 :INPUT) = [QUIT] [KWIT]
PRINTWORLD
END

TO  FRONT :OBJECT :LIST
IF  :OBJECT > COUNT :LIST [OUTPUT :LIST]
IF  :OBJECT < 1 [OUTPUT []]
OUTPUT SENTENCE FIRST :LIST FRONT
   :OBJECT - 1 BUTFIRST :LIST
END
```

The simplest task for the computer is to tell where all the blocks are. All the computer has to do is examine :WORLD row by row. DESCRIBE.WORLD handles this search, and DESCRIBE tells which blocks are in that row, starting with the top and working its way down. Recall that :WORLD contains only the first letter of each colored block. PR.COLOR prints out the entire word based on that first letter. PR.COLOR appears in other procedures as well.

```
TO  DESCRIBE.WORLD
IF  NOT :INPUT = [TELL ME WHAT YOU SEE]
   [PRINT [I DO NOT UNDERSTAND] STOP]
PRINT []
PRINT [STARTING FROM THE LEFT]
PRINT []
DESC.LOOP 1 :WORLD
END

]TO DESC.LOOP :NO :WORLD
DESCRIBE ITEM :NO :WORLD
IF :NO < 6 [DESC.LOOP :NO + 1 :WORLD]
END
```

```
TO  DESCRIBE :LIST
IF  :LIST = [] [STOP]
IF  (FIRST :LIST) = ". [PRINT [... A
   SPACE] STOP]
IF  MEMBER? LAST :LIST :COLORS
   [SUB.DESCRIBE :LIST]
DESCRIBE BUTLAST :LIST
END

TO  SUB.DESCRIBE :LIST
MAKE "COLOR LAST :LIST
PRINT (SENTENCE [IT'S THE] PR.COLOR
   [BLOCK])
IF  (COUNT :LIST) > 1 [PRINT [AND BELOW
   THAT...]]
END

TO  PR.COLOR
IF  :COLOR = "B [OUTPUT "BLUE]
IF  :COLOR = "W [OUTPUT "WHITE]
IF  :COLOR = "G [OUTPUT "GREEN]
IF  :COLOR = "R [OUTPUT "RED]
END
```

If the computer is asked to find one of the blocks, the procedure FIND.BLOCK is called. It breaks down the problem into three actions: parsing the user's sentence to see which block is to be found, finding the block, and telling what is around that block.

```
TO  FIND.BLOCK
PRINT [LOOKING]
PRINT []
FIND.PARSE
IF  NOT MEMBER? :COLOR :COLORS [PRINT [I
   DO NOT KNOW THAT COLOR] STOP]
FIND.COLUMN 1
FIND.LEVEL 1
PRINT (SENTENCE [FOUND IN COLUMN] :COLUMN
   [LEVEL] :LEVEL)
PRINT []
REPORT.FIND
END
```

FIND.PARSE has the job of learning the color of the block the user wants. Since all sentences of this type start with the words "Where is the" these words are stripped off the sentence by REMOVE.INPUT. Then :COLOR is made the first letter of the next word. If this letter does not represent one

of the known colors, the computer responds with "I do not know that color," and everything is halted. If it is a recognized color, the computer looks for the block. FIND.COLUMN and FIND.LEVEL take care of this duty.

```
TO FIND.PARSE
REMOVE.INPUT 3
MAKE "COLOR FIRST FIRST:INPUT
END

TO REMOVE.INPUT :NUM
IF :NUM > COUNT :INPUT [STOP]
IF :NUM < 1 [STOP]
MAKE "INPUT BUTFIRST :INPUT
REMOVE.INPUT :NUM - 1
END

TO FIND.COLUMN :NUM
IF MEMBER? :COLOR ITEM :NUM :WORLD [MAKE
  "COLUMN :NUM STOP]
FIND.COLUMN :NUM + 1
END

TO FIND.LEVEL :NUM
IF (ITEM :NUM ITEM :COLUMN :WORLD) =
  :COLOR [MAKE "LEVEL :NUM STOP]
FIND.LEVEL :NUM + 1
END
```

Next, the computer reports where the block is and what blocks, if any, are near it. REPORT.FIND manages the report. ABOVE.REPORT looks one space above the block and tells which block is in that location. BELOW.REPORT checks under the block. LEFT.REPORT and RIGHT.REPORT look to either side of the block and tell which blocks are there.

```
TO REPORT.FIND
ABOVE.REPORT :LEVEL
MAKE "COLOR ITEM :LEVEL ITEM :COLUMN
  :WORLD
BELOW.REPORT :LEVEL
LEFT.REPORT
RIGHT.REPORT
END

TO ABOVE.REPORT :LEVEL
IF :LEVEL > 5 [STOP]
IF (ITEM :LEVEL +1 ITEM :COLUMN :WORLD) =
  ". [PRINT [NOTHING ABOVE THAT]
  STOP]
MAKE "COLOR ITEM :LEVEL + 1 ITEM :COLUMN
  :WORLD
PRINT (SENTENCE [ABOVE THAT, I SEE THE ]
  PR.COLOR [ONE])
ABOVE.REPORT :LEVEL + 1
END

TO BELOW.REPORT :LEVEL
IF :LEVEL <2 [STOP]
PRINT (SENTENCE [BELOW THE] PR.COLOR [ONE
  IS]) .
MAKE "COLOR ITEM :LEVEL - 1 ITEM :COLUMN
  :WORLD
PRINT SENTENCE [THE] PR.COLOR
BELOW.REPORT :LEVEL - 1
END

TO LEFT.REPORT
IF :COLUMN = 1 [PRINT [NOTHING TO THE
  LEFT] STOP]
MAKE "LEFT.OBJ ITEM :LEVEL ITEM (:COLUMN
  - 1) :WORLD
IF NOT :LEFT.OBJ = ". [MAKE "COLOR
  :LEFT.OBJ
  PRINT (SENTENCE [TO THE LEFT, THE]
  PR.COLOR [ONE] ) ]
END

TO RIGHT.REPORT
IF :COLUMN = 6 [PRINT [NOTHING TO THE
  RIGHT] STOP]
MAKE "RIGHT.OBJ ITEM :LEVEL ITEM :COLUMN
  + 1 :WORLD
IF NOT :RIGHT.OBJ = ". [MAKE "COLOR
  :RIGHT.OBJ
  PRINT (SENTENCE [TO THE RIGHT, THE]
  PR.COLOR [BLOCK])]
END
```

Putting one block on top of another is a much more involved process. MOVE.BLOCKS governs the steps. First, MOVE.PARSE determines the "object," the piece to be moved, and the "target," the block on which the object will be placed. A typical sentence for moving blocks would be "Put the red one on the blue." MOVE.PARSE begins by removing the first two words; "red one on the blue" remains.

OBJECT becomes "R" for red, and the rest of the word is taken out of the sentence. If the next word is "on," two words are removed because the user didn't use a word like "block," "one," or "cube" after the color (e.g., "Put the red on the green brick"); otherwise three words are removed, as is the case here. The first letter of the first word is put into TARGET. Now OBJECT is "R" and TARGET is "B." MOVE.BLOCKS makes sure that those letters stand for known colors.

```
TO MOVE.BLOCKS
MOVE.PARSE
IF NOT MEMBER? :OBJECT :COLORS [PRINT [I
  DO NOT UNDERSTAND] STOP]
IF NOT MEMBER? :TARGET :COLORS [PRINT [I
  DO NOT UNDERSTAND] STOP]
IF :TARGET = :OBJECT [PRINT [YOU CAN'T DO
  THAT] STOP]
PRINT [MOVING]
FIND.OBJECT
CLEAR 5 :COLUMN.OBJECT :LEVEL.OBJECT
FIND.TARGET
CLEAR 5 :COLUMN.TARGET :LEVEL.TARGET
FIND.OBJECT
MOVE
END

TO MOVE.PARSE
REMOVE.INPUT 2
MAKE "OBJECT FIRST FIRST :INPUT
REMOVE.INPUT 1
IFELSE (FIRST :INPUT) = "ON
  [REMOVE.INPUT 2] [REMOVE.INPUT 3]
MAKE "TARGET FIRST FIRST :INPUT
END
```

Now the computer uses FIND.OBJECT to locate the block that needs to be moved. Sometimes there are other blocks on top of OBJECT, and they have to be cleared off. CLEAR looks to see if there are any, calls FINDSPOT to locate a new spot for each, and then moves the block out of the way. FINDSPOT is used to find an out-of-the-way place for any wayward block. It does this by selecting a random number and placing the brick at the top of that column as long as it isn't the column for :OBJECT or :TARGET. MOVEBLOCK executes the actual move.

```
TO FIND.OBJECT
MAKE "COLOR :OBJECT
FIND.COLUMN 1
MAKE "COLUMN.OBJECT :COLUMN
FIND.LEVEL 1
MAKE "LEVEL.OBJECT :LEVEL
END

TO CLEAR :NUM :COLUMN :LEVEL
IF :NUM = :LEVEL [STOP]
MAKE "BLOCK ITEM :NUM ITEM :COLUMN :WORLD
IF NOT :BLOCK = ". [MAKE "COLOR :BLOCK
  PRINT
  (SENTENCE [MOVING THE] PR.COLOR [BLOCK
  OUT OF THE WAY])
  FINDSPOT MOVEBLOCK]
CLEAR :NUM - 1 :COLUMN :LEVEL
END

TO FINDSPOT
MAKE "TEMP.COLUMN (1 + RANDOM 5)
IF OR :TEMP.COLUMN = :COLUMN.TARGET
  :TEMP.COLUMN = :COLUMN.OBJECT
  [FINDSPOT]
FINDSPOT.LEVEL 1
END

TO FINDSPOT.LEVEL :NUM
IF :NUM > 5 [STOP]
IF (ITEM :NUM ITEM :TEMP.COLUMN :WORLD) =
  ". [MAKE "TEMP.LEVEL :NUM
 STOP]
FINDSPOT.LEVEL :NUM + 1
END
TO MOVEBLOCK
TAKE.OBJECT :COLUMN :NUM
PUT.OBJECT :TEMP.COLUMN :TEMP.LEVEL
  :BLOCK
END
```

Once everything has been cleared from the "object" block, the "target" block is located using FIND.TARGET, and any blocks on top of it are cleared off. Then it is time to move the one block onto the other. This is handled by MOVE.

```
TO FIND.TARGET
MAKE "COLOR :TARGET
FIND.COLUMN 1
MAKE "COLUMN.TARGET :COLUMN
```

```
FIND.LEVEL 1
MAKE "LEVEL.TARGET :LEVEL
END

TO MOVE
PRINT ( SENTENCE [OBJECT FOUND IN COLUMN]
   :COLUMN.OBJECT
   [LEVEL] :LEVEL.OBJECT )
PRINT ( SENTENCE [TARGET FOUND IN COLUMN]
   :COLUMN.TARGET
   [LEVEL] :LEVEL.TARGET )
TAKE.OBJECT :COLUMN.OBJECT :LEVEL.OBJECT
MAKE "TEMP :LEVEL.TARGET + 1
PUT.OBJECT :COLUMN.TARGET :TEMP :OBJECT
END
```

Possible Extensions and Expansions

- Use Logo's graphics capabilities to graphically represent the Block World.
- When the program has to move a brick out of the way, nothing is shown on the screen except a message that the block is being moved. Use graphics to show each block being moved.
- Use animation to show the blocks actually move.
- Use a robot arm (maybe Fischer Technik or LEGO) to move blocks under computer control.
- Add to the vocabulary that the program can understand so that it can recognize the same requests it does now, but in different words. Instead of "Tell me what you see," perhaps the program could also understand "Where are all the blocks." Instead of "Put the Blue on the Green," perhaps "Move the Blue on the Green" could be used. There are many possibilities.
- BLOCKS can only place one block on top of another. Add other spatial relationships so there will be a greater variety of places where you could put blocks. These relationships might include under, beside, to the right, or to the left.
- Be able to move a block into an empty space. The program could choose a space randomly, as it does now when it clears off a block, or the user could describe which empty cell she wants.
- Add other properties to the blocks, as in SHRDLU. Some could be large or small, or maybe tall or flat. Perhaps some rules would have to be made that would allow you to put a small block on a large one but not the other way around.
- Add a third dimension and all the appropriate spatial relationships. You could then put a block in front of or behind another block. This would look particularly nice with graphic drawings of the world, as suggested above.

BLOCKS
THE PROGRAM
LogoWriter Version

```
TO BLOCKS
INITIALIZE
BLOCK.LOOP
END

TO INITIALIZE
MAKE "COLORS [R W B G]
MAKE "WORLD [[. . . . .] [R . . . .]
   [W . . . .] [B . . . .]
   [G . . . .] [. . . . .]]
CC CT CG HT
PRINT [WELCOME TO THE WORLD OF BLOCKS]
PRINT [] PRINT []
PRINTWORLD
END
```

```
TO PRINTWORLD
PRINT [] PRINT []
PRINTWORLD.LOOP 5
PRINT [] PRINT []
END
TO PRINTWORLD.LOOP :LEVEL
IF :LEVEL < 1 [STOP]
(PRINT (ITEM :LEVEL ITEM 1 :WORLD) (ITEM
  :LEVEL ITEM 2 :WORLD)
  (ITEM :LEVEL ITEM 3 :WORLD) (ITEM
  :LEVEL ITEM 4 :WORLD)
  (ITEM :LEVEL ITEM 5 :WORLD) (ITEM
  :LEVEL ITEM 6 :WORLD) )
PRINTWORLD.LOOP :LEVEL - 1
END

TO BLOCK.LOOP
GETUSER.INPUT
PARSE.AND.ACT
BLOCK.LOOP
END

TO GETUSER.INPUT
TYPE1 [>]
MAKE "INPUT READLISTCC
END
TO PARSE.AND.ACT
IF (FRONT 2 :INPUT) = [WHERE IS]
  [FIND.BLOCK]
IF (FRONT 3 :INPUT) = [TELL ME WHAT]
  [DESCRIBE.WORLD]
IF (FRONT 2 :INPUT) = [PUT THE]
  [MOVE.BLOCKS]
IF (FRONT 1 :INPUT) = [QUIT] [KWIT]
PRINTWORLD
END

TO FRONT :OBJECT :LIST
IF :OBJECT > COUNT :LIST [OUTPUT :LIST]
IF :OBJECT < 1 [OUTPUT []]
OUTPUT SENTENCE FIRST :LIST
  FRONT :OBJECT - 1 BUTFIRST :LIST
END
```

```
TO FIND.BLOCK
PRINT [LOOKING]
PRINT []
FIND.PARSE
IF NOT MEMBER? :COLOR :COLORS [PRINT [' I
  DO NOT KNOW THAT COLOR']
  STOP]
FIND.COLUMN 1
FIND.LEVEL 1
PRINT (SENTENCE [FOUND IN COLUMN] :COLUMN
  [LEVEL] :LEVEL)
PRINT []
REPORT.FIND
END

TO FIND.PARSE
REMOVE.INPUT 3
MAKE "COLOR FIRST FIRST :INPUT
END

TO REMOVE.INPUT :NUM
IF :NUM > COUNT :INPUT [STOP]
IF :NUM < 1 [STOP]
MAKE "INPUT BUTFIRST :INPUT
REMOVE.INPUT :NUM - 1
END

TO FIND.COLUMN :NUM
IF MEMBER? :COLOR ITEM :NUM :WORLD [MAKE
  "COLUMN :NUM STOP]
FIND.COLUMN :NUM + 1
END
TO FIND.LEVEL :NUM
IF (ITEM :NUM ITEM :COLUMN :WORLD) =
  :COLOR [MAKE "LEVEL :NUM STOP]
FIND.LEVEL :NUM + 1
END

TO REPORT.FIND
ABOVE.REPORT :LEVEL
MAKE "COLOR ITEM :LEVEL ITEM :COLUMN
  :WORLD
BELOW.REPORT :LEVEL
LEFT.REPORT
RIGHT.REPORT
END
```

```
TO ABOVE.REPORT :LEVEL
IF :LEVEL > 5 [STOP]
IF (ITEM :LEVEL +1 ITEM :COLUMN :WORLD) =
  ". [PRINT [NOTHING ABOVE THAT]
  STOP]
MAKE "COLOR ITEM :LEVEL + 1 ITEM :COLUMN
  :WORLD
PRINT (SENTENCE [ABOVE THAT, I SEE THE ]
  PR.COLOR [ONE])
ABOVE.REPORT :LEVEL + 1
END

TO PR.COLOR
IF :COLOR = "B [OUTPUT "BLUE]
IF :COLOR = "W [OUTPUT "WHITE]
IF :COLOR = "G [OUTPUT "GREEN]
IF :COLOR = "R [OUTPUT "RED]
END

TO BELOW.REPORT :LEVEL
IF :LEVEL <2 [STOP]
PRINT (SENTENCE [BELOW THE] PR.COLOR [ONE
  IS])
MAKE "COLOR ITEM :LEVEL - 1 ITEM :COLUMN
  :WORLD
PRINT SENTENCE [THE] PR.COLOR
BELOW.REPORT :LEVEL - 1
END

TO LEFT.REPORT
IF :COLUMN = 1 [PRINT [NOTHING TO THE
  LEFT] STOP]
MAKE "LEFT.OBJ ITEM :LEVEL
  ITEM (:COLUMN - 1) :WORLD
IF NOT :LEFT.OBJ = ". [MAKE "COLOR
  :LEFT.OBJ
  PRINT (SENTENCE [TO THE LEFT, THE]
  PR.COLOR [ONE] ) ]
END

TO RIGHT.REPORT
IF :COLUMN = 6 [PRINT [NOTHING TO THE
  RIGHT] STOP]
MAKE "RIGHT.OBJ ITEM :LEVEL ITEM :COLUMN
  + 1 :WORLD
IF NOT :RIGHT.OBJ = ". [MAKE "COLOR
  :RIGHT.OBJ
  PRINT (SENTENCE [TO THE RIGHT, THE]
  PR.COLOR [BLOCK])]
END

TO DESCRIBE.WORLD
IF NOT :INPUT = [TELL ME WHAT YOU SEE]
  [PRINT [I DO NOT UNDERSTAND] STOP]
PRINT []
PRINT [STARTING FROM THE LEFT]
PRINT []
DESC.LOOP 1 :WORLD
END

TO DESC.LOOP :NO :WORLD
DESCRIBE ITEM :NO :WORLD
IF :NO < 6 [DESC.LOOP :NO + 1 :WORLD]
END

TO DESCRIBE :LIST
IF :LIST = [] [STOP]
IF (FIRST :LIST) = ". [PRINT [... A
  SPACE] STOP]
IF MEMBER? LAST :LIST :COLORS
  [SUB.DESCRIBE :LIST]
DESCRIBE BUTLAST :LIST
END
TO SUB.DESCRIBE :LIST
MAKE "COLOR LAST :LIST
PRINT (SENTENCE [IT'S THE] PR.COLOR
  [BLOCK])
IF (COUNT :LIST) > 1 [PRINT [AND BELOW
  THAT...]]
END

TO MOVE.BLOCKS
MOVE.PARSE
IF NOT MEMBER? :OBJECT :COLORS [PRINT [I
  DO NOT UNDERSTAND] STOP]
IF NOT MEMBER? :TARGET :COLORS [PRINT [I
  DO NOT UNDERSTAND] STOP]
IF :TARGET = :OBJECT [PRINT [I CAN'T DO
  THAT] STOP]
PRINT [MOVING]
FIND.OBJECT
CLEAR 5 :COLUMN.OBJECT :LEVEL.OBJECT
FIND.TARGET
CLEAR 5 :COLUMN.TARGET :LEVEL.TARGET
FIND.OBJECT
MOVE
END

TO MOVE.PARSE
REMOVE.INPUT 2
MAKE "OBJECT FIRST FIRST :INPUT
REMOVE.INPUT 1
IFELSE (FIRST :INPUT) = "ON [REMOVE.INPUT
  2] [REMOVE.INPUT 3]
MAKE "TARGET FIRST FIRST :INPUT
END
```

```
TO FIND.OBJECT
MAKE "COLOR :OBJECT
FIND.COLUMN 1
MAKE "COLUMN.OBJECT :COLUMN
FIND.LEVEL 1
MAKE "LEVEL.OBJECT :LEVEL
END

TO CLEAR :NUM :COLUMN :LEVEL
IF :NUM = :LEVEL [STOP]
MAKE "BLOCK ITEM :NUM ITEM :COLUMN :WORLD
IF NOT :BLOCK = ". [MAKE "COLOR :BLOCK
  PRINT (SENTENCE [MOVING THE]
  PR.COLOR [BLOCK OUT OF THE WAY])
  FINDSPOT MOVEBLOCK]
CLEAR :NUM - 1 :COLUMN :LEVEL
END

TO FINDSPOT
MAKE "TEMP.COLUMN (1 + RANDOM 5)
IF OR :TEMP.COLUMN = :COLUMN.TARGET
  :TEMP.COLUMN = :COLUMN.OBJECT
  [FINDSPOT]
FINDSPOT.LEVEL 1

END
TO FINDSPOT.LEVEL :NUM
IF :NUM > 5 [STOP]
IF (ITEM :NUM ITEM :TEMP.COLUMN :WORLD) =
  ". [MAKE "TEMP.LEVEL :NUM STOP]
FINDSPOT.LEVEL :NUM + 1
END

TO MOVEBLOCK
TAKE.OBJECT :COLUMN :NUM
PUT.OBJECT :TEMP.COLUMN :TEMP.LEVEL
  :BLOCK
END

TO TAKE.OBJECT :COLUMN :LEVEL
SPLIT.WORLD
MAKE "COLUMN (SENTENCE :FRONT.LEVEL ".
  BUTFIRST :BACK.LEVEL)
MAKE "WORLD LPUT :COLUMN :FRONT.COLUMN
MAKE "WORLD SENTENCE :WORLD BUTFIRST
  :BACK.COLUMN
END

TO SPLIT.WORLD
MAKE "FRONT.COLUMN []
MAKE "BACK.COLUMN :WORLD
SPLIT.COLUMN 1
MAKE "FRONT.LEVEL []
MAKE "BACK.LEVEL FIRST :BACK.COLUMN
SPLIT.LEVEL 1
END

TO SPLIT.COLUMN :NUM
IF :NUM = :COLUMN [STOP]
MAKE "FRONT.COLUMN LPUT FIRST
  :BACK.COLUMN :FRONT.COLUMN
MAKE "BACK.COLUMN BUTFIRST :BACK.COLUMN
SPLIT.COLUMN: NUM + 1
END

TO SPLIT.LEVEL :NUM
IF :NUM = :LEVEL [STOP]
MAKE "FRONT.LEVEL SENTENCE :FRONT.LEVEL
  FIRST :BACK.LEVEL
MAKE "BACK.LEVEL BUTFIRST :BACK.LEVEL
SPLIT.LEVEL :NUM + 1
END

TO PUT.OBJECT :COLUMN :LEVEL :BLOCK
SPLIT.WORLD
MAKE "COLUMN ( SENTENCE :FRONT.LEVEL
  FIRST :BLOCK BUTFIRST :BACK.LEVEL )
MAKE "WORLD LPUT :COLUMN :FRONT.COLUMN
MAKE "WORLD SENTENCE :WORLD BUTFIRST
  :BACK.COLUMN
END

TO FIND.TARGET
MAKE "COLOR :TARGET
FIND.COLUMN 1
MAKE "COLUMN.TARGET :COLUMN
FIND.LEVEL 1
MAKE "LEVEL.TARGET :LEVEL
END

TO MOVE
PRINT ( SENTENCE [OBJECT FOUND IN COLUMN]
  :COLUMN.OBJECT
  [LEVEL] :LEVEL.OBJECT )
PRINT ( SENTENCE [TARGET FOUND IN COLUMN]
  :COLUMN.TARGET
  [LEVEL] :LEVEL.TARGET )
TAKE.OBJECT :COLUMN.OBJECT :LEVEL.OBJECT
MAKE "TEMP :LEVEL.TARGET + 1
PUT.OBJECT :COLUMN.TARGET :TEMP :OBJECT
END

TO KWIT
REPEAT 4 [PRINT []]
PRINT [' GOODBYE']
REPEAT 4 [PRINT []]
STOPALL
END
```

BLOCKS
THE PROGRAM
Terrapin Logo PLUS Version

```
TO BLOCKS
  INITIALIZE
  BLOCK.LOOP
END

TO INITIALIZE
 MAKE "COLORS [R W B G]
 MAKE "WORLD [[. . . . .] [R . . . .]
   [W . . . .] [B . . . .]
   [G . . . .] [. . . . .]]
  RANDOMIZE
  CLEARTEXT
 PRINT [WELCOME TO THE WORLD OF BLOCKS]
 PRINT [] PRINT []
  PRINTWORLD
END

TO PRINTWORLD
 PRINT [] PRINT []
  PRINTWORLD.LOOP 5
 PRINT [] PRINT []
END

TO PRINTWORLD.LOOP :LEVEL
 IF :LEVEL < 1 THEN STOP
 PRINT1 [' ']
 PRINT1 ITEM :LEVEL ITEM 1 :WORLD
 PRINT1 ITEM :LEVEL ITEM 2 :WORLD
 PRINT1 ITEM :LEVEL ITEM 3 :WORLD
 PRINT1 ITEM :LEVEL ITEM 4 :WORLD
 PRINT1 ITEM :LEVEL ITEM 5 :WORLD
 PRINT1 ITEM :LEVEL ITEM 6 :WORLD
 PRINT []
 PRINTWORLD.LOOP :LEVEL - 1
END

TO BLOCK.LOOP
  GETUSER.INPUT
  PARSE.AND.ACT
  BLOCK.LOOP
END

TO GETUSER.INPUT
 PRINT1 [>]
 MAKE "INPUT REQUEST
END

TO PARSE.AND.ACT
 IF FRONT 2 :INPUT = [WHERE IS] THEN
  FIND.BLOCK
 IF FRONT 3 :INPUT = [TELL ME WHAT] THEN
  DESCRIBE.WORLD
 IF FRONT 2 :INPUT = [PUT THE] THEN
  MOVE.BLOCKS
 IF FRONT 1 :INPUT = [QUIT] THEN KWIT
  PRINTWORLD
END

TO FRONT :OBJECT :LIST
 IF :OBJECT > COUNT :LIST THEN OUTPUT
  :LIST
 IF :OBJECT < 1 THEN OUTPUT []
 OUTPUT SENTENCE FIRST :LIST FRONT
  :OBJECT - 1 BUTFIRST :LIST
END

TO FIND.BLOCK
 PRINT [LOOKING]
 PRINT []
  FIND.PARSE
 IF NOT MEMBER? :COLOR :COLORS THEN PRINT
  [' I DO NOT KNOW THAT COLOR']
 STOP
 FIND.COLUMN 1
 FIND.LEVEL 1
 PRINT (SENTENCE [FOUND IN COLUMN]
  :COLUMN [LEVEL] :LEVEL)
 PRINT []
  REPORT.FIND
END

TO FIND.PARSE
 REMOVE.INPUT 3
 MAKE "COLOR FIRST FIRST :INPUT
END

TO REMOVE.INPUT :NUM
 IF :NUM > COUNT :INPUT THEN STOP
 IF :NUM < 1 THEN STOP
 MAKE "INPUT BUTFIRST :INPUT
 REMOVE.INPUT :NUM - 1
END

TO FIND.COLUMN :NUM
 IF MEMBER? :COLOR ITEM :NUM :WORLD THEN
  MAKE "COLUMN :NUM STOP
 FIND.COLUMN :NUM + 1
END
```

```
TO FIND.LEVEL :NUM
 IF ITEM :NUM ITEM :COLUMN :WORLD =
  :COLOR THEN MAKE "LEVEL :NUM STOP
 FIND.LEVEL :NUM + 1
END

TO REPORT.FIND
 ABOVE.REPORT :LEVEL
 MAKE "COLOR ITEM :LEVEL ITEM :COLUMN
  :WORLD
 BELOW.REPORT :LEVEL
 LEFT.REPORT
 RIGHT.REPORT
END

TO ABOVE.REPORT :LEVEL
 IF :LEVEL > 5 THEN STOP
 IF ITEM :LEVEL +1 ITEM :COLUMN :WORLD =
  ". THEN PRINT [NOTHING ABOVE THAT]
 STOP
 MAKE "COLOR ITEM :LEVEL + 1 ITEM :COLUMN
  :WORLD
 PRINT (SENTENCE [ABOVE THAT, I SEE THE ]
  PR.COLOR [ONE])
 ABOVE.REPORT :LEVEL + 1
END

TO PR.COLOR
 IF :COLOR = "B THEN OUTPUT "BLUE
 IF :COLOR = "W THEN OUTPUT "WHITE
 IF :COLOR = "G THEN OUTPUT "GREEN
 IF :COLOR = "R THEN OUTPUT "RED
END

TO BELOW.REPORT :LEVEL
 IF :LEVEL <2 THEN STOP
 PRINT1 (SENTENCE [BELOW THE] PR.COLOR
  [ONE IS])
 MAKE "COLOR ITEM :LEVEL - 1 ITEM :COLUMN
  :WORLD
 PRINT SENTENCE [' THE'] PR.COLOR
 BELOW.REPORT :LEVEL - 1
END

TO LEFT.REPORT
 IF :COLUMN = 1 THEN PRINT [NOTHING TO
  THE LEFT] STOP
 MAKE "LEFT.OBJ ITEM :LEVEL ITEM
  :COLUMN - 1 :WORLD
 TEST :LEFT.OBJ = ".
 IFF MAKE "COLOR :LEFT.OBJ
 IFF PRINT (SENTENCE [TO THE LEFT, THE]
  PR.COLOR [ONE])
END

TO RIGHT.REPORT
 IF :COLUMN = 6 THEN PRINT [NOTHING TO
  THE RIGHT] STOP
 MAKE "RIGHT.OBJ ITEM :LEVEL ITEM :COLUMN
  + 1 :WORLD
 TEST :RIGHT.OBJ = ".
 IFF MAKE "COLOR :RIGHT.OBJ
 IFF PRINT (SENTENCE [TO THE RIGHT, THE]
  PR.COLOR [BLOCK])
END

TO DESCRIBE.WORLD
 IF NOT :INPUT = [TELL ME WHAT YOU SEE]
  THEN PRINT [' I DO NOT UNDERSTAND']
  STOP
 PRINT []
 PRINT [STARTING FROM THE LEFT]
 PRINT []
 DESC.LOOP 1 :WORLD
END

TO DESC.LOOP :NO :WORLD
 DESCRIBE ITEM :NO :WORLD
 IF :NO < 6 THEN DESC.LOOP :NO + 1 :WORLD
END

TO DESCRIBE :LIST
 IF :LIST = [] THEN STOP
 IF FIRST :LIST = ". THEN PRINT [... A
  SPACE] STOP
 TEST MEMBER? LAST :LIST :COLORS
 IFT MAKE "COLOR LAST :LIST
 IFT PRINT (SENTENCE [IT'S THE] PR.COLOR
  [BLOCK])
 IFT IF COUNT :LIST > 1 THEN PRINT [AND
  BELOW THAT...]
 DESCRIBE BUTLAST :LIST
END
```

```
TO MOVE.BLOCKS
 MOVE.PARSE
 IF NOT MEMBER? :OBJECT :COLORS THEN
  PRINT [' I DO NOT UNDERSTAND']
 STOP
 IF NOT MEMBER? :TARGET :COLORS THEN
  PRINT [' I DO NOT UNDERSTAND']
 STOP
 IF :TARGET = :OBJECT THEN PRINT [I CAN'T
  DO THAT] STOP
 PRINT [MOVING]
 FIND.OBJECT
 CLEAR 5 :COLUMN.OBJECT :LEVEL.OBJECT
 FIND.TARGET
 CLEAR 5 :COLUMN.TARGET :LEVEL.TARGET
 FIND.OBJECT
 MOVE
END

TO MOVE.PARSE
 REMOVE.INPUT 2
 MAKE "OBJECT FIRST FIRST :INPUT
 REMOVE.INPUT 1
 TEST FIRST :INPUT = "ON
 IFT REMOVE.INPUT 2
 IFF REMOVE.INPUT 3
 MAKE "TARGET FIRST FIRST :INPUT
END

TO FIND.OBJECT
 MAKE "COLOR :OBJECT
 FIND.COLUMN 1
 MAKE "COLUMN.OBJECT :COLUMN
 FIND.LEVEL 1
 MAKE "LEVEL.OBJECT :LEVEL
END

TO CLEAR :NUM :COLUMN :LEVEL
 IF :NUM = :LEVEL THEN STOP
 MAKE "BLOCK ITEM :NUM ITEM :COLUMN
  :WORLD
 TEST NOT :BLOCK = ".
 IFT MAKE "COLOR :BLOCK
 IFT PRINT (SENTENCE [MOVING THE]
  PR.COLOR [BLOCK OUT OF THE WAY])
 IFT FINDSPOT MOVEBLOCK
 CLEAR :NUM - 1 :COLUMN :LEVEL
END

TO FINDSPOT
 MAKE "TEMP.COLUMN (1 + RANDOM 5)
 IF ANYOF :TEMP.COLUMN = :COLUMN.TARGET
  :TEMP.COLUMN = :COLUMN.OBJECT
 THEN FINDSPOT
 FINDSPOT.LEVEL 1
END

TO FINDSPOT.LEVEL :NUM
 IF :NUM > 5 THEN STOP
 IF ITEM :NUM ITEM :TEMP.COLUMN :WORLD =
  ". THEN MAKE "TEMP.LEVEL :NUM STOP
 FINDSPOT.LEVEL :NUM + 1
END

TO MOVEBLOCK
 TAKE.OBJECT :COLUMN :NUM
 PUT.OBJECT :TEMP.COLUMN :TEMP.LEVEL
  :BLOCK
END

TO TAKE.OBJECT :COLUMN :LEVEL
 SPLIT.WORLD
 MAKE "COLUMN (SENTENCE :FRONT.LEVEL ".
  BUTFIRST :BACK.LEVEL)
 MAKE "WORLD LPUT :COLUMN :FRONT.COLUMN
 MAKE "WORLD SENTENCE :WORLD BUTFIRST
  :BACK.COLUMN
END

TO SPLIT.WORLD
 MAKE "FRONT.COLUMN []
 MAKE "BACK.COLUMN :WORLD
 SPLIT.COLUMN 1
 MAKE "FRONT.LEVEL []
 MAKE "BACK.LEVEL FIRST :BACK.COLUMN
 SPLIT.LEVEL 1
END

TO SPLIT.COLUMN :NUM
 IF :NUM = :COLUMN THEN STOP
 MAKE "FRONT.COLUMN LPUT FIRST
  :BACK.COLUMN :FRONT.COLUMN
 MAKE "BACK.COLUMN BUTFIRST :BACK.COLUMN
 SPLIT.COLUMN :NUM + 1
END

TO SPLIT.LEVEL :NUM
 IF :NUM = :LEVEL THEN STOP
 MAKE "FRONT.LEVEL SENTENCE :FRONT.LEVEL
  FIRST :BACK.LEVEL
 MAKE "BACK.LEVEL BUTFIRST :BACK.LEVEL
 SPLIT.LEVEL :NUM + 1
END
```

```
TO PUT.OBJECT :COLUMN :LEVEL :BLOCK
  SPLIT.WORLD
  MAKE "COLUMN ( SENTENCE :FRONT.LEVEL
   FIRST :BLOCK BUTFIRST :BACK.LEVEL )
  MAKE "WORLD LPUT :COLUMN :FRONT.COLUMN
  MAKE "WORLD SENTENCE :WORLD BUTFIRST
   :BACK.COLUMN
END

TO FIND.TARGET
  MAKE "COLOR :TARGET
  FIND.COLUMN 1
  MAKE "COLUMN.TARGET :COLUMN
  FIND.LEVEL 1
  MAKE "LEVEL.TARGET :LEVEL
END

TO MOVE
  PRINT ( SENTENCE [OBJECT FOUND IN
   COLUMN] :COLUMN.OBJECT
   [LEVEL] :LEVEL.OBJECT )
  PRINT ( SENTENCE [TARGET FOUND IN
   COLUMN] :COLUMN.TARGET
   [LEVEL] :LEVEL.TARGET )
  TAKE.OBJECT :COLUMN.OBJECT :LEVEL.OBJECT
  MAKE "TEMP :LEVEL.TARGET + 1
  PUT.OBJECT :COLUMN.TARGET :TEMP :OBJECT
END

TO KWIT
  REPEAT 4 [PRINT []]
  PRINT [' GOODBYE']
  REPEAT 4 [PRINT []]
  TOPLEVEL
END
```

6 INFER— COMPUTERS AND LOGICAL INFERENCE

Many types of AI programs require a portion of their programming to be some kind of inference engine. An inference engine is the part of a program that applies some rules or logic to known data to give a desirable result. The part of our SMARTTT program that evaluates and selects from different moves when playing for the computer is in general terms an inference engine. Expert systems (to be discussed in more depth in an upcoming chapter) rely heavily on some sort of inference engine.

CYRUS is an expert system developed at Yale University. It has been "fed" extensive information about the lives of former Secretaries of State, Cyrus Vance and Edmund Muskie, and can answer questions related to their lives and work. CYRUS's inference engine produces some interesting results in the following conversation about Cyrus Vance:

USER: When was the last time you were in Egypt?

CYRUS: On Dec. 10, 1978.

USER: Why did you go there?

CYRUS: To negotiate the Camp David accords.

USER: Who did you talk to there?

CYRUS: With Anwar Sadat.

USER: Has your wife ever met Mrs. Begin?

CYRUS: Yes, most recently at a state dinner in Israel in Jan. 1980.

The amazing part is that CYRUS was never told that the two wives had ever met! Since CYRUS knew Mrs. Begin usually accompanied her husband to state dinners and since it knew that Mrs. Vance went on that particular trip to Israel, it deduced that the two wives had met (Rosenthal, 1983).

Our inference engine deals with two areas of artificial intelligence: learning and deductive reasoning. The user types statements, and the program builds a knowledge base that is used later when the user quizzes the computer. To understand how the program works, one must understand a little about logic and reasoning.

Logic concerns itself with arguments, validity, and truth. An argument is made up of several statements, called premises, and a conclusion based on the premises. An argument is valid if, assuming that the premises are true, the conclusion of the argument is also true. It is beyond the scope of a computer program to determine truth, but a computer can be programmed to use valid logic.

This program uses one valid rule of reasoning called the Hypothetical Syllogism. The classic example goes like this:

SOCRATES IS A MAN
ALL MEN ARE MORTAL
THEREFORE SOCRATES
IS MORTAL

A syllogism takes two or more premises and forms a conclusion. Syllogisms are all of the following form: A is a B, B is a C, therefore A is a C. In the example above, the premises are SOCRATES IS A MAN and ALL MEN ARE MORTAL. The conclusion is

SOCRATES IS MORTAL. This form of logical reasoning will produce a true conclusion as long as the premises are true. Consider the following example:

```
MY CAR IS A BIRD
ALL BIRDS HAVE FEATHERS
THEREFORE MY CAR HAS
FEATHERS
```

The conclusion is false, of course, but if my car *were* a bird it would indeed have feathers. It is a false conclusion only because a premise is false. A false premise can also produce a true conclusion:

```
MY CAT IS A MAN
ALL MEN ARE MORTAL
THEREFORE MY CAT IS MORTAL
```

Validity guarantees only that the conclusion is true given that the premises are true.

The Program

Programs that can process syllogisms appear in several texts (Hartnell, 1985; Krutch, 1986). The Inference Engine is our Logo program that stores information, deduces answers to questions (using syllogisms), and recalls data on request. The user "converses" with the computer in English, even if the program understands only a very limited grammar. The program must assume that the sentences are true, since it cannot determine the truth of each statement. As seen in the examples above, false sentences can produce very interesting conclusions.

Below is an example of one conversation with the inference engine. The program uses an asterisk (*) as a prompt for the user's input. Note that the user does not use punctuation to end a statement. Although the user would use all capital letters when running the program, in the examples below, the uppercase statements are the program's responses and the mixed case are the user's.

```
* Rover is a dog
   UNDERSTOOD
* A dog is a mammal
   UNDERSTOOD
* A mammal is a warm-blooded creature
   UNDERSTOOD
* Is Rover a mammal
   THINKING
   YES
* Rover is a friend
   UNDERSTOOD
* A dog is Rover
   UNDERSTOOD
* A dog is a four-legged animal
   UNDERSTOOD
* Explain Rover
   ROVER IS:
   DOG
   FRIEND
```

The knowledge base is the key to INFER's abilities. Called :DATA, the knowledge base is a lists of lists. The first item in each sublist is the topic (:NOUN1) for that list, and each other item is something known about that topic (from all the :NOUN2s). :DATA for the dog example above is:

```
[ [ [dog] [mammal] [Rover] [four-legged
  animal] ]
[ [Rover] [dog] [friend] ]
[ [mammal] [warm-blooded creature] ] ]
```

An additional list called :TOPICS is used to keep track of the topics currently known to :DATA. :TOPICS is used to simplify the process of finding out what :DATA knows and what it doesn't. For this example, :TOPICS is

```
[ [mammal] [dog] [Rover] ]
```

The program understands declarative sentences, inquiries, and requests for information. Each of the three types of statements is in a different form, and the computer recognizes the type from its form. The declarative statement tells the computer information to add to its data bank. Examples of declaratives include: "Rover is a Dog" and "A mammal is a warm-blooded creature." All declarative sentences are of the following form:

```
Noun phrase "Is" Noun phrase
```

The program recognizes a declarative by the word "is" in the middle.

```
TO ANALYZE.DECLARE
DECLARE.PARSE
LEARN
(PR :INPUT [- UNDERSTOOD])
END

TO DECLARE.PARSE
MAKE "NOUN1 FIRST :INPUT
MAKE "NOUN2 BUTFIRST :INPUT
SUB.D.PARSE
REMOVE.ART.NOUN1
REMOVE.ART.NOUN2
IF WORD? :NOUN1 [MAKE "NOUN1 SENTENCE
  :NOUN1 []]
IF WORD? :NOUN2 [MAKE "NOUN2 SENTENCE
  :NOUN2 []]
END

TO SUB.D.PARSE
IF (FIRST :NOUN2) = "IS [MAKE "NOUN2
  BUTFIRST :NOUN2 STOP]
MAKE "NOUN1 SENTENCE :NOUN1 FIRST :NOUN2
MAKE "NOUN2 BUTFIRST :NOUN2
SUB.D.PARSE
END

TO REMOVE.ART.NOUN1
IF MEMBER? FIRST :NOUN1 :ARTICLES [MAKE
  "NOUN1 BUTFIRST :NOUN1]
END

TO REMOVE.ART.NOUN2
IF MEMBER? FIRST :NOUN2 :ARTICLES [MAKE
  "NOUN2 BUTFIRST :NOUN2]
END
```

The procedure that handles declarative sentences is called ANALYZE.DECLARE. It works in two parts; the first is the parser, and the second stores the data in the knowledge base. The parser breaks the sentence into the first noun phrase, :NOUN1, and the second noun phrase, :NOUN2. In the process, any leading articles (an, a, the) are removed from the two noun phrases by REMOVE.ART.NOUN1 and REMOVE.ART.NOUN2. For example, if you typed "A computer is a machine," NOUN1 would become [computer] and NOUN2 would become [machine].

```
TO LEARN
IFELSE MEMBER? :NOUN1 :TOPICS
  [EXPAND.KNOWLEDGE] [ADD.TO.DATA]
END

TO ADD.TO.DATA
MAKE "TOPICS FPUT :NOUN1 :TOPICS
MAKE "TEMPLIST LIST :NOUN1 :NOUN2
MAKE "DATA FPUT :TEMPLIST :DATA
END

TO EXPAND.KNOWLEDGE
FINDLIST
MAKE "TEMPLIST LPUT :NOUN2 :TEMPLIST
MAKE "DATA FPUT :TEMPLIST SENTENCE :LIST1
  :LIST2
END
```

Next, the program adds the new information to :DATA. First the procedure looks to see if [computer] is already in :DATA. If it is, EXPAND.KNOWLEDGE is called and [machine] is added to the information already known about [computer]. If [computer] is not already known to :DATA, ADD.TO.DATA is called, putting [computer] and [machine] into the knowledge base.

If the user were to type "A dog is a pet," since [dog] is already in :DATA, EXPAND.KNOWLEDGE would add [pet] to the list and :DATA would look something like this:

```
[ [ [dog] [mammal] [Rover] [four-legged
  animal] [pet] ]
[ [Rover] [dog] [friend] ]
[ [mammal] [warm-blooded creature] ] ]
```

:TOPICS would be unchanged since [dog] is already in the list. On the other hand, if the user were to then type "A fish is a pet," the program would react differently using ADD.TO.DATA. First, [fish] would be added to :TOPICS, then [fish] and [pet] would be added to :DATA. Notice that it doesn't matter that [pet] has been used before. :DATA now looks like this:

```
[ [ [fish] [pet] ]
[ [dog] [mammal] [Rover] [four-legged
  animal] [pet] ]
[ [Rover] [dog] [friend] ]
[ [mammal] [warm-blooded creature] ] ]
```

Inquiries are by far the most interesting part of the inference engine from an artificial intelligence perspective. The user can ask questions for which the program must deduce an answer. The user can ask "Is Rover a four-legged animal" and the computer will respond "YES" even though the computer was never explicitly told that Rover had four legs. All inquiries are of the form

```
"IS" Noun phrase Article Noun phrase
```

Inquiries are recognized because they begin with the word "is." Driven by ANALYZE.INQUIRY the parser, INQUIRY.PARSER, removes the first word and then proceeds to break the remaining part of the sentence into the two noun phrases. The indicator of the break between the two is the article. There *must* be an article separating the two noun phrases. When I first ran a prototype of the program, I typed "Rover is my friend" and then "Is Rover my friend." The program failed since it ran out of sentence before it found the article. One solution to this problem you may want to later implement is to add possessive pronouns to the article list. I have some reservations about this, however, since "my friend" certainly carries a different connotation than "a friend."

```
TO ANALYZE.INQUIRY
PR [THINKING]
INQUIRY.PARSE
SUB.I.ANALYZE
END

TO INQUIRY.PARSE
MAKE "NOUN1 FIRST BUTFIRST :INPUT
MAKE "NOUN2 BUTFIRST BUTFIRST :INPUT
SUB.I.PARSE
REMOVE.ART.NOUN1
REMOVE.ART.NOUN2
IF WORD? :NOUN1 [MAKE "NOUN1 SENTENCE
  :NOUN1 []]
IF WORD? :NOUN2 [MAKE "NOUN2 SENTENCE
  :NOUN2 []]
END

TO SUB.I.PARSE
IF MEMBER? FIRST :NOUN2 :ARTICLES [MAKE
  "NOUN2 BUTFIRST :NOUN2 STOP]
MAKE "NOUN1 SENTENCE :NOUN1 FIRST :NOUN2
MAKE "NOUN2 BUTFIRST :NOUN2
SUB.I.PARSE
END
```

Once the two noun phrases are isolated in :NOUN1 and :NOUN2, SUB.I.ANALYZE checks to make sure that :NOUN1 is in :TOPICS. If it is not, the computer responds much as it did to a request for data for a topic not known to the computer. If :NOUN1 is known to :DATA, the computer first finds the sublist containing all the things the user directly told the computer on that topic. If :NOUN2 is in the list, the computer has a simple task and answers "YES" without having to do any deducing. If the user typed "Is a dog a mammal," the computer would respond with "YES" since the computer was told that directly.

If :NOUN2 is not in the list, then the user never directly told the computer the answer to the question. The program must try to infer the answer. From this point, there are only two responses that the

computer uses: either "YES" or "I DO NOT KNOW." The computer has not been programmed to establish if the answer is "NO," although some rules of logic, if implemented, could sometimes offer "NO" as the correct answer.

To infer, the computer must draw connections between the topic and those things known about it. Suppose the user wrote "Is Rover a warm-blooded creature." A human with the same knowledge as :DATA would say "Well, Rover is a dog, and dogs are mammals, and mammals are warm-blooded creatures, so the answer is yes, Rover is a warm-blooded creature." The computer must search for those connections, however.

```
TO SUB.I.ANALYZE
IF NOT MEMBER? :NOUN1 :TOPICS
  [PR SENTENCE [I DON'T KNOW ANYTHING
  ABOUT] :NOUN1 STOP]
FINDLIST
IF MEMBER? :NOUN2 :TEMPLIST [(PR :INPUT
  [- YES]) STOP]
MAKE "DEDUCED "FALSE
MAKE "NOUN :NOUN1
MAKE "SUBLIST BUTFIRST :TEMPLIST
DEDUCE
IF NOT :DEDUCED [PR [' I DO NOT KNOW']]
END

TO DEDUCE
IF :SUBLIST = [] [STOP]
MAKE "NOUN1 FIRST :SUBLIST
IF AND (NOT :NOUN1 = :NOUN2) MEMBER?
  :NOUN1 :TOPICS [SUB.DEDUCE]
MAKE "SUBLIST BUTFIRST :SUBLIST
DEDUCE
END

TO SUB.DEDUCE
FINDLIST
IF MEMBER? :NOUN2 :TEMPLIST [(PR :INPUT
  [-YES])
  MAKE "DEDUCED "TRUE STOP]
MAKE "TEMPLIST BUTFIRST :TEMPLIST
BUILDSUBLIST
END

TO BUILDSUBLIST
IF :TEMPLIST = [] [STOP]
MAKE "SUBLIST LPUT FIRST :TEMPLIST
  :SUBLIST
MAKE "TEMPLIST BUTFIRST :TEMPLIST
BUILDSUBLIST
END
```

The inference engine uses a inference algorithm of ingenious design. SUB.I.ANALYZE starts by taking the list it already has of everything directly known on the topic and places it in :SUBLIST. In our example above, :SUBLIST would be

```
[ [dog] [friend] ]
```

DEDUCE is then called to look at each item in :SUBLIST to see what it knows, trying to draw the connection. As it works, :SUBLIST grows as additional connections are discovered.

DEDUCE checks to see if it knows anything about the first thing in :SUBLIST. If it does, it checks to see if :NOUN2 is in that list. For example, [dog] is the first thing in SUBLIST, and since [dog] is in :TOPICS, the computer finds the list in :DATA for [dog]. It looks like this:

```
[ [dog] [mammal] [Rover] [four-legged
  animal] [pet] ].
```

The procedure checks to see if :NOUN2 is part of that list. Since [warm-blooded creature] isn't there, all of the things in that list (except [dog], since we just checked it) are added to :SUBLIST as possible future connections to check. The procedure BUILDSUBLIST does this. The first item in :SUBLIST is also removed since we are through with the check. :SUBLIST now looks like this:

```
[ [friend] [mammal] [Rover] [four-legged
  animal] [pet] ].
```

The procedure then repeats the steps. This time, since the computer doesn't know anything about [friend], the program just removes the first item in the list, and :SUBLIST now looks like this:

```
[ [mammal] [Rover] [four-legged animal]
  [pet] ].
```

This time through, the computer finds [warm-blooded creature] in the list of things about mammals and thus answers "YES," then stops deducing. If the computer goes through all of :SUBLIST and finds no connection, it answers "I DONT KNOW."

In addition, DEDUCE and its subprocedures check to see that the first item in :SUBLIST is not the original topic. If it were and the program were to add its list to :SUBLIST, the program could get caught in an infinite loop of checking and rechecking the same connections over and over again.

Requests for known information is the third type of statement the inference engine understands. Recall in the sample dialog with INFER that the user typed "Explain Rover" and the computer responded with

ROVER IS:
DOG
FRIEND

A user can request all the known information on a topic by using the word "explain." All requests are of the form

```
"Explain"  Noun  phrase
```

ANALYZE.REQUEST simply removes the first word, and the remaining noun phrase is placed in :NOUN1. The program looks for the sublist whose topic is that in :NOUN1, then prints out the things in that list one by one using PRINTLIST. First, of course, the program makes sure that :NOUN1 is in :TOPICS. If it is not, the computer responds with "I KNOW NOTHING ABOUT ..." If the user were to type "Explain a horse" in our example, the computer would reply "I KNOW NOTHING ABOUT HORSE."

```
TO  ANALYZE.REQUEST
MAKE  "NOUN1  BUTFIRST  :INPUT
REMOVE.ART.NOUN1
IF  NOT  MEMBER?  :NOUN1  :TOPICS  [PR
   SENTENCE  [I  HAVE  NO  DATA  ON]  :NOUN1
   STOP]
FINDLIST
PR  SENTENCE  :NOUN1  [IS:]
PRINTLIST  :TEMPLIST
END

TO  PRINTLIST  :TEMPLIST
IF  :TEMPLIST  =  []  [STOP]
PRINT  FIRST  :TEMPLIST
PRINTLIST  :TEMPLIST
END
```

INFER calls INITIALIZE to set up the initial value of some global variables, as well as other preparation tasks, such as clearing the screen and printing a welcome message. The meat of the program is driven by a procedure called INFER.DRIVER. This procedure first gets the user's sentence. It then looks to see if the input is one of the three types of sentences the program understands. If it is not one of the three, the computer responds, "I DONT UNDERSTAND."

```
TO  INFER
INITIALIZE
INFER.DRIVER
END

TO  INITIALIZE
CG
HT
CT
CC
PR  []
PR  [WELCOME  TO  THE  INFERENCE  ENGINE]
PR  []
MAKE  "ARTICLES  [A  AN  THE]
MAKE  "DATA  []
MAKE  "TOPICS  []
END
```

```
TO  INFER.DRIVER
GET.SENTENCE
IF  INQUIRY?  [ANALYZE.INQUIRY]
IF  EXPLAIN?  [ANALYZE.REQUEST]
IF  DECLARE?  [ANALYZE.DECLARE]
IF  QUIT?  [KWIT]
IF  NONE.OF.ABOVE  [OTHERWISE.ANALYZE]
INFER.DRIVER
END
```

Possible Extensions and Expansions

- The program could also check to see if some of the declarations are duplicates. As the program now stands, if the user types "a dog is a mammal" twice, [mammal] will appear in the list for [dog] twice. This may cause problems in an inquiry.
- Remove punctuation from the end of any sentence. That would remove the concern of a program crash that accompanies the urge to put a question mark at the end of an inquiry.
- Another modification might be added to the inquiry process itself. If the computer answers yes to an inquiry, that data could be added to the data list for that topic. For example, "Is Rover a warm-blooded creature" was a rather lengthy deduction but once the computer deduced the answer, the computer could "learn" it. [warm-blooded creature] could be added to the list for [Rover], and if the question were raised again, the computer would only need to check its memory and not have to do any deductive reasoning.
- Allow for the saving of :DATA so that when the program is next used, the user could build on the last conversation.
- Have DEDUCE check to see if it has already examined a path, e.g., don't check "dog" twice.
- Program INFER to correct false knowledge learned. If someone types in "A dog is a cat" provide a command so that the user can correct the error.
- Form checking could be added to the parser so that if the computer thought the user's sentence were a familiar form and then on parsing discovered that it did not fit the form, the program wouldn't fail but would rather give a message like "I don't understand."
- Have the computer answer inquiries that are not separated by an article.
- Further rules of logic could be programmed.

INFER
THE PROGRAM
LogoWriter Version

```
TO  INFER
INITIALIZE
INFER.DRIVER
END

TO  INITIALIZE
CG
HT
CT
CC
PR []
PR [WELCOME TO THE INFERENCE ENGINE]
PR []
MAKE "ARTICLES [A AN THE]
MAKE "DATA []
MAKE "TOPICS []
END

TO  INFER.DRIVER
GET.SENTENCE
IF  INQUIRY?  [ANALYZE.INQUIRY]
IF  EXPLAIN?  [ANALYZE.REQUEST]
IF  DECLARE?  [ANALYZE.DECLARE]
IF  QUIT?  [KWIT]
IF  NONE.OF.ABOVE  [OTHERWISE.ANALYZE]
INFER.DRIVER
END

TO  GET.SENTENCE
TYPE "*
MAKE "INPUT  READLISTCC
END
```

```
TO NONE.OF.ABOVE
OUTPUT NOT (OR INQUIRY? QUIT? EXPLAIN?
  DECLARE?)
END

TO OTHERWISE.ANALYZE
PR [I DON'T UNDERSTAND.]
END

TO QUIT?
IF (FIRST :INPUT) = "QUIT [OUTPUT "TRUE]
OUTPUT "FALSE
END

TO KWIT
REPEAT 4 [PR []]
PR [GOODBYE]
REPEAT 4 [PR []]
STOPALL
END

TO DECLARE?
IF AND MEMBER? "IS :INPUT NOT INQUIRY?
  [OUTPUT "TRUE]
OUTPUT "FALSE
END

TO ANALYZE.DECLARE
DECLARE.PARSE
LEARN
(PR :INPUT [- UNDERSTOOD])
END

TO DECLARE.PARSE
MAKE "NOUN1 FIRST :INPUT
MAKE "NOUN2 BUTFIRST :INPUT
SUB.D.PARSE
REMOVE.ART.NOUN1
REMOVE.ART.NOUN2
IF WORD? :NOUN1 [MAKE "NOUN1 SENTENCE
  :NOUN1 []]
IF WORD? :NOUN2 [MAKE "NOUN2 SENTENCE
  :NOUN2 []]
END

TO SUB.D.PARSE
IF (FIRST :NOUN2) = "IS [MAKE "NOUN2
  BUTFIRST :NOUN2 STOP]
MAKE "NOUN1 SENTENCE :NOUN1 FIRST :NOUN2
MAKE "NOUN2 BUTFIRST :NOUN2
SUB.D.PARSE
END

TO LEARN
IFELSE MEMBER? :NOUN1 :TOPICS
  [EXPAND.KNOWLEDGE] [ADD.TO.DATA]
END

TO ADD.TO.DATA
MAKE "TOPICS FPUT :NOUN1 :TOPICS
MAKE "TEMPLIST LIST :NOUN1 :NOUN2
MAKE "DATA FPUT :TEMPLIST :DATA
END

TO EXPAND.KNOWLEDGE
FINDLIST
MAKE "TEMPLIST LPUT :NOUN2 :TEMPLIST
MAKE "DATA FPUT :TEMPLIST SENTENCE :LIST1
  :LIST2
END

TO INQUIRY?
IF (FIRST :INPUT) = "IS [OUTPUT "TRUE]
OUTPUT "FALSE
END

TO ANALYZE.INQUIRY
PR [THINKING]
INQUIRY.PARSE
SUB.I.ANALYZE
END

TO INQUIRY.PARSE
MAKE "NOUN1 FIRST BUTFIRST :INPUT
MAKE "NOUN2 BUTFIRST BUTFIRST :INPUT
SUB.I.PARSE
REMOVE.ART.NOUN1
REMOVE.ART.NOUN2
IF WORD? :NOUN1 [MAKE "NOUN1 SENTENCE
  :NOUN1 []]
IF WORD? :NOUN2 [MAKE "NOUN2 SENTENCE
  :NOUN2 []]
END

TO SUB.I.PARSE
IF MEMBER? FIRST :NOUN2 :ARTICLES [MAKE
  "NOUN2 BUTFIRST :NOUN2 STOP]
MAKE "NOUN1 SENTENCE :NOUN1 FIRST :NOUN2
MAKE "NOUN2 BUTFIRST :NOUN2
SUB.I.PARSE
END
```

```
TO  SUB.I.ANALYZE
IF  NOT  MEMBER?  :NOUN1  :TOPICS
   [PR  SENTENCE  [I  DON'T  KNOW  ANYTHING
   ABOUT]  :NOUN1  STOP]
FINDLIST
IF  MEMBER?  :NOUN2  :TEMPLIST  [(PR  :INPUT
   [-  YES])  STOP]
MAKE  "DEDUCED  "FALSE
MAKE  "NOUN  :NOUN1
MAKE  "SUBLIST  BUTFIRST  :TEMPLIST
DEDUCE
IF  NOT  :DEDUCED  [PR  [' I DO NOT KNOW']]
END

TO  DEDUCE
IF  :SUBLIST  =  []  [STOP]
MAKE  "NOUN1  FIRST  :SUBLIST
IF  AND  (NOT  :NOUN1  =  :NOUN2)  MEMBER?
   :NOUN1  :TOPICS  [SUB.DEDUCE]
MAKE  "SUBLIST  BUTFIRST  :SUBLIST
DEDUCE
END

TO  SUB.DEDUCE
FINDLIST
IF  MEMBER?  :NOUN2  :TEMPLIST  [(PR  :INPUT
   [-YES])  MAKE  "DEDUCED  "TRUE  STOP]
MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
BUILDSUBLIST
END

TO  BUILDSUBLIST
IF  :TEMPLIST  =  []  [STOP]
MAKE  "SUBLIST  LPUT  FIRST  :TEMPLIST
   :SUBLIST
MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
BUILDSUBLIST
END

TO  EXPLAIN?
IF  (FIRST  :INPUT)  =  "EXPLAIN  [OUTPUT
   "TRUE]
OUTPUT  "FALSE
END

TO  ANALYZE.REQUEST
MAKE  "NOUN1  BUTFIRST  :INPUT
REMOVE.ART.NOUN1
IF  NOT  MEMBER?  :NOUN1  :TOPICS
   [PR  SENTENCE  [I  HAVE  NO  DATA  ON]
   :NOUN1  STOP]
FINDLIST
PR  SENTENCE  :NOUN1  [IS:]
PRINTLIST  :TEMPLIST
END

TO  PRINTLIST  :TEMPLIST
MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
IF  :TEMPLIST  =  []  [STOP]
PRINT  FIRST  :TEMPLIST
PRINTLIST  :TEMPLIST
END

TO  REMOVE.ART.NOUN1
IF  MEMBER?  FIRST  :NOUN1  :ARTICLES  [MAKE
   "NOUN1  BUTFIRST  :NOUN1]
END

TO  REMOVE.ART.NOUN2
IF  MEMBER?  FIRST  :NOUN2  :ARTICLES  [MAKE
   "NOUN2  BUTFIRST  :NOUN2]
END

TO  FINDLIST
MAKE  "LIST1  []
MAKE  "LIST2  :DATA
SUBFINDLIST
END

TO  SUBFINDLIST
IF  (FIRST  FIRST  :LIST2)  =  :NOUN1
   [MAKE  "TEMPLIST  FIRST  :LIST2  MAKE
   "LIST2  BUTFIRST  :LIST2  STOP]
MAKE  "LIST1  LPUT  FIRST  :LIST2  :LIST1
MAKE  "LIST2  BUTFIRST  :LIST2
SUBFINDLIST
END
```

INFER
THE PROGRAM
Terrapin Logo PLUS Version

```
TO INFER
  INITIALIZE
  INFER.DRIVER
END

TO INITIALIZE
  CLEARTEXT
 PR []
 PR [WELCOME TO THE INFERENCE ENGINE]
 PR []
 MAKE "ARTICLES [A AN THE]
 MAKE "DATA []
 MAKE "TOPICS []
END

TO INFER.DRIVER
  GET.SENTENCE
  IF INQUIRY? THEN ANALYZE.INQUIRY
  IF EXPLAIN? THEN ANALYZE.REQUEST
  IF DECLARE? THEN ANALYZE.DECLARE
  IF QUIT? THEN KWIT
  IF NONE.OF.ABOVE THEN OTHERWISE.ANALYZE
  INFER.DRIVER
END

TO GET.SENTENCE
 PRINT1 "*
 MAKE "INPUT REQUEST
END

TO NONE.OF.ABOVE
 OUTPUT NOT (ANYOF INQUIRY? QUIT?
  EXPLAIN? DECLARE?)
END

TO OTHERWISE.ANALYZE
 PR [I DON'T UNDERSTAND.]
END

TO QUIT?
 IF FIRST :INPUT = "QUIT THEN OUTPUT
  "TRUE
 OUTPUT "FALSE
END

TO KWIT
 REPEAT 4 [PR []]
 PR [' GOODBYE']
 REPEAT 4 [PR []]
  TOPLEVEL
END

TO DECLARE?
 IF ALLOF MEMBER? "IS :INPUT NOT INQUIRY?
  OUTPUT "TRUE
 OUTPUT "FALSE
END

TO ANALYZE.DECLARE
 DECLARE.PARSE
 LEARN
 PR [' UNDERSTOOD']
END

TO DECLARE.PARSE
 MAKE "NOUN1 FIRST :INPUT
 MAKE "NOUN2 BUTFIRST :INPUT
  SUB.D.PARSE
  REMOVE.ART.NOUN1
  REMOVE.ART.NOUN2
 IF WORD? :NOUN1 THEN MAKE "NOUN1
  SENTENCE :NOUN1 []
 IF WORD? :NOUN2 THEN MAKE "NOUN2
  SENTENCE :NOUN2 []
END

TO SUB.D.PARSE
 IF FIRST :NOUN2 = "IS THEN MAKE "NOUN2
  BUTFIRST :NOUN2 STOP
 MAKE "NOUN1 SENTENCE :NOUN1 FIRST :NOUN2
 MAKE "NOUN2 BUTFIRST :NOUN2
  SUB.D.PARSE
END

TO LEARN
 TEST MEMBER? :NOUN1 :TOPICS
  IFFALSE ADD.TO.DATA
  IFTRUE EXPAND.KNOWLEDGE
END

TO ADD.TO.DATA
 MAKE "TOPICS FPUT :NOUN1 :TOPICS
 MAKE "TEMPLIST LIST :NOUN1 :NOUN2
 MAKE "DATA FPUT :TEMPLIST :DATA
END
```

```
TO  EXPAND.KNOWLEDGE
  FINDLIST
  MAKE  "TEMPLIST  LPUT  :NOUN2  :TEMPLIST
  MAKE  "DATA  FPUT  :TEMPLIST  SENTENCE
   :LIST1  :LIST2
END

TO  INQUIRY?
  IF  FIRST  :INPUT  =  "IS  THEN  OUTPUT  "TRUE
  OUTPUT  "FALSE
END

TO  ANALYZE.INQUIRY
  PR  [' THINKING']
   INQUIRY.PARSE
   SUB.E.ANALYZE
END

TO  INQUIRY.PARSE
  MAKE  "NOUN1  FIRST  BUTFIRST  :INPUT
  MAKE  "NOUN2  BUTFIRST  BUTFIRST  :INPUT
   SUB.I.PARSE
   REMOVE.ART.NOUN1
   REMOVE.ART.NOUN2
   IF  WORD?  :NOUN1  THEN  MAKE  "NOUN1
    SENTENCE  :NOUN1  []
   IF  WORD?  :NOUN2  THEN  MAKE  "NOUN2
    SENTENCE  :NOUN2  []
END

TO  SUB.I.PARSE
  IF  MEMBER?  FIRST  :NOUN2  :ARTICLES  THEN
   MAKE  "NOUN2  BUTFIRST  :NOUN2  STOP
  MAKE  "NOUN1  SENTENCE  :NOUN1  FIRST  :NOUN2
  MAKE  "NOUN2  BUTFIRST  :NOUN2
   SUB.I.PARSE
END

TO  SUB.I.ANALYZE
  IF  NOT  MEMBER?  :NOUN1  :TOPICS  THEN
  PR  SENTENCE  [I  DON'T  KNOW  ANYTHING
   ABOUT]  :NOUN1  STOP
  FINDLIST
  IF  MEMBER?  :NOUN2  :TEMPLIST  THEN  PR
   [YES]  STOP
  MAKE  "DEDUCED  "FALSE
  MAKE  "NOUN  :NOUN1
  MAKE  "SUBLIST  BUTFIRST  :TEMPLIST
  DEDUCE
  IF  NOT  :DEDUCED  THEN  PR  [' I  DO  NOT
   KNOW']
END

TO  DEDUCE
  IF  :SUBLIST  =  []  THEN  STOP
  MAKE  "NOUN1  FIRST  :SUBLIST
  TEST  ALLOF  (NOT  :NOUN1  =  :NOUN2)  MEMBER?
   :NOUN1  :TOPICS
  IFT  FINDLIST
  IFT  IF  MEMBER?  :NOUN2  :TEMPLIST  THEN  PR
   [' YES']
  MAKE  "DEDUCED  "TRUE  STOP
  IFT  MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
  IFT  BUILDSUBLIST
  MAKE  "SUBLIST  BUTFIRST  :SUBLIST
  DEDUCE
END

TO  BUILDSUBLIST
  IF  :TEMPLIST  =  []  THEN  STOP
  MAKE  "SUBLIST  LPUT  FIRST  :TEMPLIST
   :SUBLIST
  MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
  BUILDSUBLIST
END

TO  EXPLAIN?
  IF  FIRST  :INPUT  =  "EXPLAIN  THEN  OUTPUT
   "TRUE
  OUTPUT  "FALSE
END

TO  ANALYZE.REQUEST
  MAKE  "NOUN1  BUTFIRST  :INPUT
  REMOVE.ART.NOUN1
  IF  NOT  MEMBER?  :NOUN1  :TOPICS  THEN
  PR  SENTENCE  [I  HAVE  NO  DATA  ON]  :NOUN1
   STOP
  FINDLIST
  PR  SENTENCE  :NOUN1  [IS:]
   PRINTLIST  :TEMPLIST
END

TO  PRINTLIST  :TEMPLIST
  MAKE  "TEMPLIST  BUTFIRST  :TEMPLIST
  IF  :TEMPLIST  =  []  THEN  STOP
  PRINT  SENTENCE  [' '] FIRST  :TEMPLIST
   PRINTLIST  :TEMPLIST
END

TO  REMOVE.ART.NOUN1
  IF  MEMBER?  FIRST  :NOUN1  :ARTICLES  THEN
   MAKE  "NOUN1  BUTFIRST  :NOUN1
END
```

```
TO  REMOVE.ART.NOUN2
  IF  MEMBER?  FIRST  :NOUN2  :ARTICLES  THEN
   MAKE  "NOUN2  BUTFIRST  :NOUN2
END

TO  FINDLIST
 MAKE  "LIST1  []
 MAKE  "LIST2  :DATA
  SUBFINDLIST
END

TO  SUBFINDLIST
 TEST  FIRST  FIRST  :LIST2  =  :NOUN1
 IFT  MAKE  "TEMPLIST  FIRST  :LIST2  MAKE
  "LIST2  BUTFIRST  :LIST2  STOP
 MAKE  "LIST1  LPUT  FIRST  :LIST2  :LIST1
 MAKE  "LIST2  BUTFIRST  :LIST2
  SUBFINDLIST
END
```

7 SUPER.INFER— KNOWLEDGE REPRESENTATIONS IN INTELLIGENT PROGRAMS

Our last program, INFER, has already laid much of the groundwork for our next two chapters. In this chapter, we examine one way of representing knowledge within an intelligent system. In the next chapter, we will investigate grammars and create one to be used with our improved version of INFER.

We've created a few knowledge bases, but let's discuss the beginnings of a more general approach to building a knowledge base. Knowledge can be thought of as a collection of facts, but those facts are meaningless in isolation. It is the interconnectedness of facts that give them meaning. The human mind is capable of retaining millions of facts and all their accompanying connections, and there are thousands of types of connections. These connections facilitate understanding and often elicit further memories. The sight of a forest can remind us of a favorite camping trip. Other connections are then made, and we think of roasting marshmallows, hikes in the woods, getting wet in the rain, the chill of the morning, and the smoky smell of the crackling fire.

Information about objects, people, concepts, and their connections is not difficult to represent in a computer program, but many attempts, including INFER, are simple "hacks" to make a program functional. Although these solutions may work well in one application, they are perhaps not good representations in another.

Ross Quillian was one of the first computer scientists to realize that knowledge representation in computer programs needed to be addressed as a problem in and of itself (Chabris 1987). Quillian's work introduced the formal idea of semantic networks. A semantic network is a chain, or system of "nodes" interconnected by links. A set of rules that gives meaning to the patterns of nodes and links is also necessary. Well designed semantic networks are good for analyzing and answering questions about the ideas in those networks.

INFER's memory was capable of processing one type of connection: classes and instances. Recall a portion of the conversation with INFER used in the last chapter:

* Rover is a dog
  UNDERSTOOD
* A dog is a mammal
  UNDERSTOOD
* A mammal is a warm-blooded creature
  UNDERSTOOD

Rover is an instance of a dog. Dog is a subclass of mammal and mammal is a sub-class of warm-blooded creature.

If we had adopted a semantic network for INFER, in our example above, Rover, dog, and mammal would be three nodes. Rover would be linked to dog and dog would be linked to mammal. The link is a one-way link since not all dogs are Rover and not all mammals are dogs. The rule that gives the network meaning is "class/instance." Rover is an instance of the class "dog," and dog is an instance of the class "mammal."

One of the reasons semantic networks are useful is that when they are represented graphically, they are easily understood, even by someone not familiar with the idea of semantic networks. The diagram for our example is on page 82.

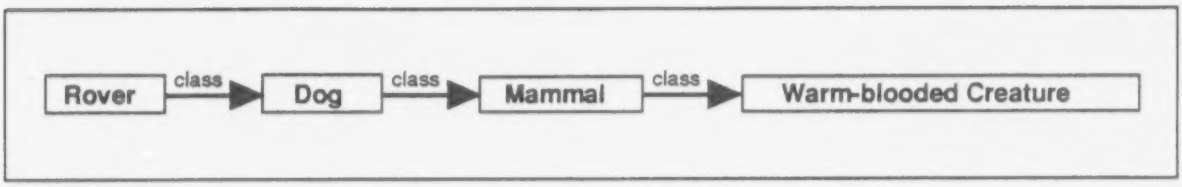


DIAGRAM 1

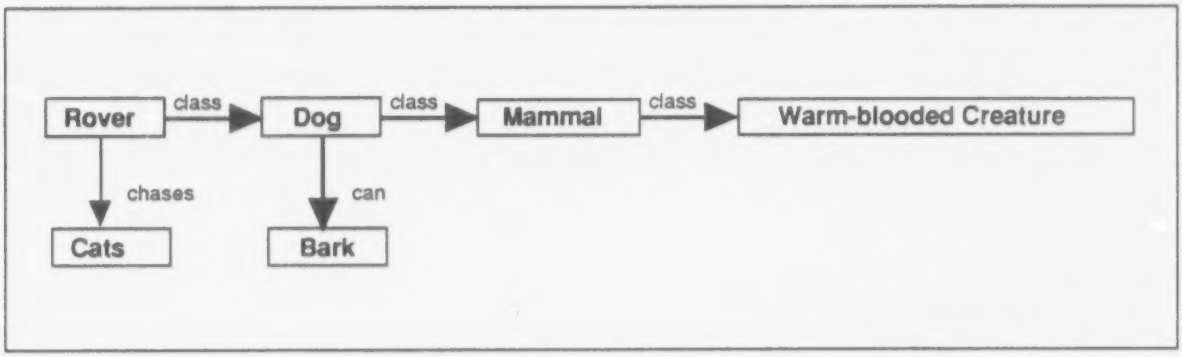


DIAGRAM 2

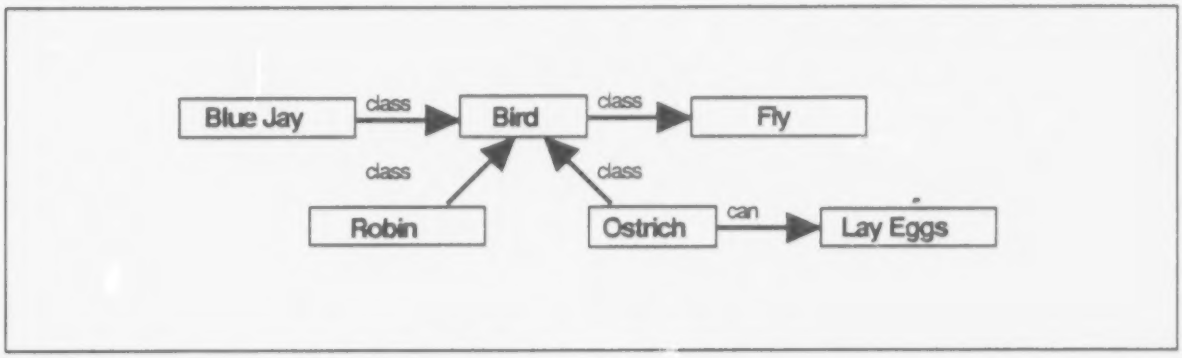


DIAGRAM 3

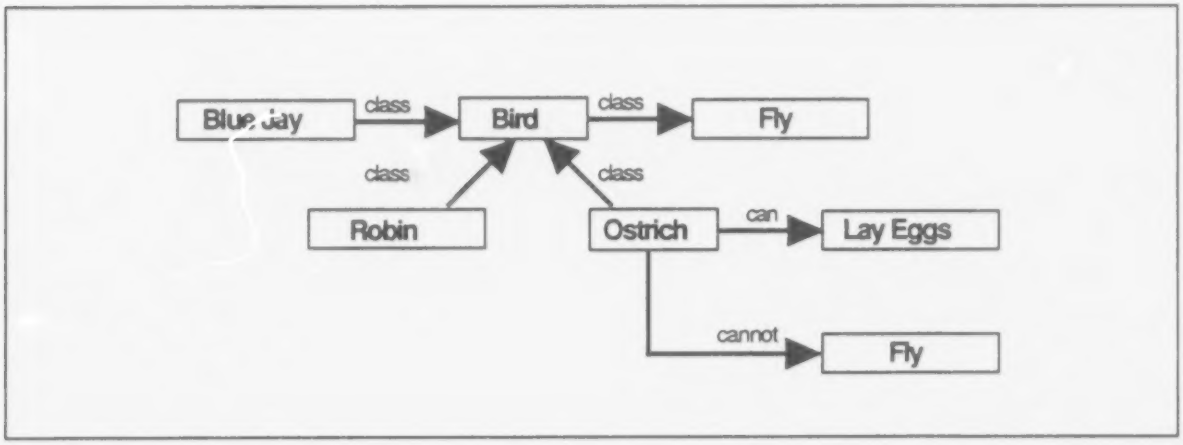


DIAGRAM 4

Other types of connections can also be made. Properties are one of the most useful types of connections. For instance, if the connection is "chases," it is easy to expand the diagram to include the idea "Rover chases cats." The idea "dogs can bark" is just as easy to incorporate. See diagram 2 on page 82.

One of the advantages of using a semantic network is that each subclass "inherits" all the properties of its superclasses. Since dogs can bark and since Rover is a dog, Rover "inherits" the ability to bark. In the last chapter we saw that Rover is a warm blooded creature even though we didn't tell the computer. Rover inherited the superclass "mammal" from "dog" and "warm-blooded creature" from the superclass "mammal." This value inheritance makes semantic networks a versatile knowledge representation.

Occasionally, however, we don't want a subclass to inherit all the properties of its connected classes. For example, consider birds. It is likely that in a conversation with the computer you may type "all birds can fly." The discussion may include some of the following sentences as well: "a robin is a bird," "an ostrich is a bird," "an ostrich can lay eggs," and "a blue jay is a bird." When asked, "Can a robin fly?" the computer would think and respond correctly "yes," but it would also respond affirmatively when asked, "Can an ostrich fly?" This is called the Token/Type Distinction Problem. See diagram 3 on page 82.

The way that many programs supporting semantic networks function is to first look at ostrich and see if the computer knows anything about whether ostriches can fly. That is, the computer looks for a link of the "can" type and sees if the node at the other end is "fly." There is one "can" link from ostrich but the node is "lay eggs." A semantic network is a data structure for use in systems that construct their own knowledge. Connections are not always made at their most logical points and information is often left out. Notice that "ostrich" has a "can" connection to "lay eggs." This connection may more logically belong off of "bird," but connections are made as knowledge is constructed. If this were a planned network, then it would likely be "logical." Since nothing to help answer the "can fly" question is found at "ostrich," the computer looks at the superclass "bird." There it finds a "can" link with the node "fly" and answers with a false "yes." This is not an error of logic but of incomplete information.

This algorithm for answering questions will work well with semantic networks as long as we provide the knowledge base with all the facts. The addition of the idea "ostriches cannot fly" adds the link "cannot" and the node "fly" to our network. See diagram 4 on page 82.

Now if the computer were asked "Can ostriches fly?" it would see the link "cannot" with the node "fly" and correctly respond "no."

How the Program Works

SUPER.INFER is an example of a program designed to process information in semantic networks. SUPER.INFER handles three types of actions. It will process a declaration, that is, teach the computer new facts. It will process an inquiry, a request for information from the knowledge base. It will also respond to a request for all the information on one object. The procedure ACT determines which action to take based on which command the user enters when asked by GET.INFO.

In order to carry out an action, SUPER.INFER needs to know the object on which the action will be performed. The program also needs to know which action

to carry out. There are three other pieces of information that the program needs to know from time to time: the class, the property, and the value of the property. GET.INFO gets each of these five pieces of information from the user.

We have usually used the procedure GET.INFO to refer to the natural language processing part of a program. This is not the case in SUPER.INFER. This program will focus on knowledge representation. We use the name GET.INFO here to make things easier in the next chapter. There we will more closely examine language processors and develop one for the program we are working on now.

```
TO GET.INFO
PRINT [WHAT IS OBJECT]
MAKE "OBJECT READLIST
PRINT [WHAT IS CLASS]
MAKE "CLASS READLIST
PRINT [WHAT IS PROPERTY]
MAKE "PROPERTY READLIST
PRINT [WHAT IS VALUE]
MAKE "VALUE READLIST
PRINT [WHAT IS ACTION]
MAKE "ACTION READLIST
END

TO CHECK.FORM
IF NOT LIST? :OBJECT [ MAKE "OBJECT
  SENTENCE :OBJECT [] ]
IF NOT LIST? :CLASS [ MAKE "CLASS
  SENTENCE :CLASS [] ]
IF NOT LIST? :PROPERTY [ MAKE "PROPERTY
  SENTENCE :PROPERTY [] ]
IF NOT LIST? :VALUE [ MAKE "VALUE
  SENTENCE :VALUE [] ]
MAKE "OBJECT REMOVE.QUANT :OBJECT
MAKE "CLASS REMOVE.QUANT :CLASS
MAKE "PROPERTY REMOVE.QUANT :PROPERTY
MAKE "VALUE REMOVE.QUANT :VALUE
END

TO REMOVE.QUANT :LIST
IF :LIST = [] [ OUTPUT :LIST ]
IF MEMBER? FIRST :LIST :QUANTIFIERS
  [ OUTPUT BUTFIRST :LIST ]
OUTPUT :LIST
END
TO ACT
IF :ACTION = [D1] [ DECLARE1 STOP ]
IF :ACTION = [D2] [ DECLARE2 STOP ]
IF :ACTION = [I1] [ INQUIRY1 STOP ]
IF :ACTION = [I2] [ INQUIRY2 STOP ]
IF :ACTION = [I3] [ INQUIRY3 STOP ]
IF :ACTION = [R1] [ REQUEST1 STOP ]
IF :ACTION = [Q1] [ KWIT ]
PRINT [I DON'T UNDERSTAND]
END
```

After GET.INFO is called, CHECK.FORM is put into action. First it makes sure that each of the five special pieces of information is a list. Next, each list is examined, and quantifiers (like "a," "my," "all," etc.) are removed. These two activities ensure that all the information is of a consistent form for the next part of the program.

DECLARE1 and DECLARE2 establish the links between nodes in the knowledge base. DECLARE1 is like DECLARE in INFER. It establishes the class connection between objects. For example, it would be called to make the connection between Rover and dog and between dog and mammal. To activate DECLARE1, the user would type in an object and a class (Rover and dog, for example), press Return for property and value since they do not apply to this procedure, and enter D1 as the action.

DECLARE2 is used to make the connection between an object and one of its properties. If we wanted the computer to know that Rover was tan and white, we would enter Rover for object, nothing for class, color for property, and tan and white for value. The action would be D2.

Besides the fact that SUPER.INFER understands properties, there is a major difference between SUPER.INFER and INFER. INFER used a list to store all of its knowledge. When the program wanted to "remember" something that it had been taught, it had to look through the entire list. People don't tend to think that way.

Our memory is more directly accessible. When you want to think of something, you think of the name of the object or a picture of the object and lots of connected ideas flow into your mind's eye. This will probably trigger other ideas, and their connected thoughts and ideas will flow forward. SUPER.INFER tries to implement this kind of content-addressable memory. Take a look at the procedures:

```
TO DECLARE1
MAKE "OBJECT MAKE.NAME :OBJECT
IFELSE NAME? :OBJECT [ MAKE :OBJECT LPUT
  :CLASS THING :OBJECT ]
  [ MAKE :OBJECT LPUT :CLASS [[]] ]
PRINT [-UNDERSTOOD]
END

TO DECLARE2
MAKE "OBJECT MAKE.NAME :OBJECT
IFELSE NAME? OBJECT [ SUB.DECLARE2]
  [ MAKE :OBJECT FPUT (FPUT :PROPERTY
  []) [] ]
MAKE "PROPERTY MAKE.NAME :PROPERTY
MAKE "NAME (WORD :OBJECT ". :PROPERTY)
IFELSE NAME? :NAME [ MAKE :NAME LPUT
  :VALUE THING :NAME ]
  [ MAKE :NAME LPUT :VALUE [] ]
PRINT [-UNDERSTOOD]
END

TO SUB.DECLARE2
IF NOT MEMBER? :PROPERTY FIRST THING
  :OBJECT
  [ MAKE :OBJECT FPUT (LPUT :PROPERTY
  FIRST THING :OBJECT)
  BUTFIRST THING :OBJECT]
END
```

As an example, think of the ideas "big dogs are vicious animals" and "big dogs' color is black." The first idea would be processed by DECLARE1. We cannot actually enter these two sentences, but by answering each of GET.INFO's questions we can give the computer all the necessary information. :OBJECT would be "big dogs" and :CLASS would be "vicious animals." :ACTION would be "D1." Since variables must have single word names, MAKE. NAME adds a period between words making :OBJECT "big.dogs." Each object in the knowledge base is a variable containing its related properties and classes. The primitive THING? checks to see if there already is a variable by a given name. Let's assume this is the first time that the computer has heard about big dogs. DECLARE1 would create a new variable called :BIG.DOGS and put [vicious animals] at the end of the list. The empty list in the front of :BIG.DOGS is for any properties that the computer may learn about later. :BIG.DOGS now looks like this:

```
[ [] [vicious animals] ]
```

To teach the computer the next idea, we make :OBJECT "big dogs" again but make :PROPERTY "color," :VALUE "black," and :ACTION "D2." DECLARE2 sees that :BIG.DOGS already exists and adds "color" to its list of properties. Then the procedure makes another variable called :BIG.DOGS.COLOR and puts [black] into it. If the computer already knew something about the color of big dogs, then it would simply add [black] to that list. :BIG.DOGS now looks like this:

```
[ [ [color] ] [vicious animals] ]
```

And :BIG.DOGS.COLOR looks like this:

```
[ [black] ]
```

We could then ask the computer to tell us everything it knows about big dogs. When asked by GET.INFO we would say :OBJECT was "big dogs" and then that :ACTION was "R1." REQUEST1 finds the node (variable) containing all the information about big dogs and looks through it, reporting to the user. It would respond:

```
BIG.DOGS IS
  VICIOUS ANIMALS
BIG.DOGS COLOR IS
  BLACK
```

91

REQUEST1 does this all with the help of PRINT.LIST and PRINT.PROPS.

```
TO  REQUEST1
MAKE  "OBJECT  MAKE.NAME  :OBJECT
IF  NOT  NAME?  :OBJECT  [  PRINT  SENTENCE  [I
  KNOW  NOTHING  ABOUT]  :OBJECT  STOP  ]
PRINT  SENTENCE  :OBJECT  "IS
PRINT.LIST  BUTFIRST  THING  :OBJECT
PRINT.PROPS  FIRST  THING  :OBJECT
END

TO  PRINT.LIST  :LIST
IF  :LIST  =  []  [  STOP  ]
(PRINT  CHAR  32  CHAR  32  FIRST  :LIST)
PRINT.LIST  BUTFIRST  :LIST
END

TO  PRINT.PROPS  :LIST
IF  :LIST  =  []  [  STOP  ]
MAKE  "PROPERTY  MAKE.NAME  FIRST  :LIST
MAKE  "NAME  (WORD  :OBJECT  ".  :PROPERTY)
PRINT  (SENTENCE  :OBJECT  FIRST  :LIST  "IS  )
PRINT.LIST  THING  :NAME
PRINT.PROPS  BUTFIRST  :LIST
END
```

To further our example, let's assume that we taught the computer that vicious animals are a public threat and that vicious animals' temperament is unruly. Now we can have the computer do some logical deduction. INQUIRY1 is like the query system in INFER. This procedure would be called for the kind of question that is looking for a class connection, such as "Are big dogs a public threat?" Again, :OBJECT would be "big dogs," :CLASS would be "public threat," and :ACTION would be "E1."

INQUIRY1 calls DEDUCE1, which creates a list of all of :BIG.DOGS superclasses and searches each for the class "public threat." If it has to, it adds the classes of the superclasses to the list and examines all the connections, however long, until it either runs out of connections or finds the one it is looking for. Like INFER, SUPER.INFER will only respond with "yes" or "I don't know."

```
TO  INQUIRY1
PRINT  [-THINKING]
IF  NOT  NAME?  MAKE.NAME  :OBJECT
   [  PRINT  SENTENCE  [I  DON'T  KNOW
   ANYTHING  ABOUT]  :OBJECT  STOP  ]
MAKE  "DEDUCED  "FALSE
DEDUCE1  BUTFIRST  THING  MAKE.NAME  :OBJECT
IF  NOT  :DEDUCED  [  PRINT  [-I  DO  NOT  KNOW]
END

TO  DEDUCE1  :DEDUCELIST
IF  :DEDUCELIST  =  []  [  STOP  ]
IF  MEMBER?  :CLASS  :DEDUCELIST  [  PRINT  [-
  YES]  MAKE  "DEDUCED  "TRUE  STOP  ]
MAKE  "CLASS1  FIRST  :DEDUCELIST
IF  AND  (NOT  :CLASS1  =  :OBJECT)  (NAME?
  MAKE.NAME  :CLASS1)
   [  MAKE  "DEDUCELIST
   SENTENCE  :DEDUCELIST  BUTFIRST  THING
   MAKE.NAME  :CLASS1  ]
DEDUCE1  BUTFIRST  :DEDUCELIST
END
```

INQUIRY2 functions very much like INQUIRY1, except that it looks through superclasses trying to find a property. If we wanted to ask the computer "Is big dogs' temperament unruly?" INQUIRY2 would look for a variable called :BIG.DOGS.TEMPERAMENT. Since there is none, the procedure would call DEDUCE2, which would examine each of big dogs' superclasses trying to see if any of them trigger "a memory" about temperament. It would find that big dogs are vicious animals and vicious animals have an unruly temperament. The computer would answer "yes."

This program makes the assumption that if something is known about a property but it does not match the property value currently being examined, the answer to the inquiry is "no." For example, if we asked the computer "Is big dogs' temperament vicious?" INQUIRY2 and DEDUCE2 would find :VICIOUS.ANIMALS.TEMPERAMENT but "vicious" wouldn't be there. The computer would answer "no." This may be a fallacy. Perhaps a better response would be "I'm not

sure" (see the Extensions section below).

```
TO  INQUIRY2
PRINT  [-THINKING]
IF  NOT  NAME?  MAKE.NAME  :OBJECT
   [  PRINT  SENTENCE  [I  DON'T  KNOW
   ANYTHING  ABOUT]  :OBJECT  STOP  ]
MAKE  "NAME  (WORD  MAKE.NAME  :OBJECT  ".
   MAKE.NAME  :PROPERTY)
IF  NAME?  :NAME  [  IFELSE  MEMBER?  :VALUE
   THING  :NAME
   [  PRINT  [-YES]  STOP  ]  [  PRINT  [-NO]
   STOP  ]  ]
MAKE  "DEDUCED  "FALSE
DEDUCE2  BUTFIRST  THING  MAKE.NAME  :OBJECT
IF  NOT  :DEDUCED  [  PRINT  [-I  DO  NOT  KNOW]]
END

TO  DEDUCE2  :DEDUCELIST
IF  :DEDUCELIST  =  []  [  STOP  ]
MAKE  "CLASS1  FIRST  :DEDUCELIST
IF  AND  (  NOT  :CLASS1  =  :OBJECT)  (NAME?
   MAKE.NAME  :CLASS1)  [SUB.DEDUCE2]
DEDUCE2  BUTFIRST  :DEDUCELIST
END

TO  SUB.DEDUCE2
MAKE  "NAME  (WORD  MAKE.NAME  :CLASS1  ".
   MAKE.NAME  :PROPERTY)
IF  NAME?  :NAME  [  IFELSE  MEMBER?  :VALUE
   THING  :NAME
   [  PRINT  [-YES]  ]  [  PRINT  [-NO]  ]  MAKE
   "DEDUCED  "TRUE  STOP  ]
MAKE  "DEDUCELIST  SENTENCE  :DEDUCELIST
   BUTFIRST  THING  MAKE.NAME  :CLASS1
END
```

The third type of inquiry searches the knowledge base for the value of a given property. We may want the computer to tell us what the temperament of big dogs is. We would tell the computer that :OBJECT is "big dogs" and that :PROPERTY is "temperament." You would make :ACTION E3. Like INQUIRY2, INQUIRY3 would call DEDUCE3 since there is no variable :BIG.DOGS.TEMPERAMENT. These procedures would examine big dogs' superclasses in search of the property "temperament." When it finds it, it lists all the property values associated with it. In this example, it would respond:

```
THE  TEMPERAMENT  OF  BIG.DOGS  IS
     UNRULY
```

If the computer cannot find any information about a property, it will answer with "I don't know."

```
TO  INQUIRY3
PRINT  [-THINKING]
IF  NOT  NAME?  MAKE.NAME  :OBJECT
   [  PRINT  SENTENCE  [I  DON'T  KNOW
   ANYTHING  ABOUT]  :OBJECT  STOP  ]
MAKE  "NAME  (WORD  MAKE.NAME  :OBJECT  ".
   MAKE.NAME  :PROPERTY)
IF  NAME?  :NAME  [  SUB.INQUIRY3  STOP  ]
MAKE  "DEDUCE  "FALSE
DEDUCE3  BUTFIRST  THING  MAKE.NAME  :OBJECT
IF  NOT  :DEDUCED  [  PRINT  [' I  DO  NOT
   KNOW']  ]
END

TO  SUB.INQUIRY3
PRINT  (SENTENCE  "THE  :PROPERTY  "OF
   :OBJECT  "IS  )
PRINT.LIST  THING  :NAME
END

TO  DEDUCE3  :DEDUCELIST
IF  :DEDUCELIST  =  []  [  STOP  ]
MAKE  "CLASS1  FIRST  :DEDUCELIST
IF  NOT  (AND  (NOT  :CLASS1  =  :OBJECT)
   (NAME?  MAKE.NAME  :CLASS1))
   [  DEDUCE3  BUTFIRST  :DEDUCELIST  STOP  ]
MAKE  "NAME  (WORD  MAKE.NAME  :CLASS1  ".
   MAKE.NAME  :PROPERTY)
IF  NAME?  :NAME  [  SUB.DEDUCE3  STOP  ]
MAKE  "DEDUCELIST  SENTENCE  :DEDUCELIST
   BUTFIRST  THING  MAKE.NAME  :CLASS1
DEDUCE3  BUTFIRST  :DEDUCELIST
END

TO  SUB.DEDUCE3
PRINT  (SENTENCE  "THE  :PROPERTY  "OF
   :OBJECT  "IS  )
PRINT.LIST  THING  :NAME
MAKE  "DEDUCED  "TRUE
END
```

Possible Extensions and Expansions

- Many of the suggested extensions for INFER apply to this program as well.
- When REQUEST1 or INQUIRY3 is called and the classes and properties are printed out, compound objects like "big dogs" are printed as BIG.DOGS. The words were concatenated so that they could be used as a variable name. It doesn't look very good, however, during the conversation with the computer. Have the computer break apart such objects so that they print as separate words.
- Have the output, especially for REQUEST1 and INQUIRY3, print more in the form of a sentence.
- It was mentioned above that sometimes the computer answered "no" to an inquiry when "no" may not be an appropriate response. This could be corrected simply by having the computer say "I'm not sure," or "I don't think so but I'm not positive." Another approach would be to have the computer ask questions about the property values it does find. It could ask, for example, "Is a vicious temperament pretty much the same as an unruly temperament?" and if the user responded affirmatively then the computer can answer the question with "yes."
- Program the computer to recognize "not" to mean "the opposite of." In the discussion of semantic networks, we mentioned that some systems would answer "no" to the question "can ostriches fly" because they recognize "cannot fly" as the opposite of "can fly." SUPER.INFER is not currently capable of doing this. (Hint: this extension would really only affect INQUIRY2 and DEDUCE2.)
- Build other types of connections into SUPER.INFER. One connection might be "can," which would tell what abilities the object would have. Another connection could be "have," which would show what the object possesses. You could limit this to objects but it could be intangibles as well. These kinds of changes would probably require modifications of GET.INFO, as well as the creation of new types of declaratives, inquiries, and a slight restructuring of the knowledge base.

SUPER.INFER
THE PROGRAM
LogoWriter Version

```
TO SUPER.INFER
INITIALIZE
INFER.DRIVER
END

TO INITIALIZE
CT CC CG HT
PRINT []
PRINT [WELCOME TO SUPER.INFER]
PRINT []
MAKE "QUANTIFIERS [A AN THE MY YOUR ALL
  ONE]
END

TO INFER.DRIVER
GET.INFO
CHECK.FORM
ACT
INFER.DRIVER
END

TO GET.INFO
PRINT [WHAT IS OBJECT]
MAKE "OBJECT READLIST
PRINT [WHAT IS CLASS]
MAKE "CLASS READLIST
PRINT [WHAT IS PROPERTY]
MAKE "PROPERTY READLIST
PRINT [WHAT IS VALUE]
MAKE "VALUE READLIST
PRINT [WHAT IS ACTION]
MAKE "ACTION READLIST
END

TO CHECK.FORM
IF NOT LIST? :OBJECT [ MAKE "OBJECT
  SENTENCE :OBJECT [] ]
IF NOT LIST? :CLASS [ MAKE "CLASS
  SENTENCE :CLASS [] ]
IF NOT LIST? :PROPERTY [ MAKE "PROPERTY
  SENTENCE :PROPERTY [] ]
IF NOT LIST? :VALUE [ MAKE "VALUE
  SENTENCE :VALUE [] ]
MAKE "OBJECT REMOVE.QUANT :OBJECT
MAKE "CLASS REMOVE.QUANT :CLASS
MAKE "PROPERTY REMOVE.QUANT :PROPERTY
MAKE "VALUE REMOVE.QUANT :VALUE
END

TO REMOVE.QUANT :LIST
IF :LIST = [] [ OUTPUT :LIST ]
IF MEMBER? FIRST :LIST :QUANTIFIERS
  [ OUTPUT BUTFIRST :LIST ]
OUTPUT :LIST
END

TO ACT
IF :ACTION = [D1] [ DECLARE1 STOP ]
IF :ACTION = [D2] [ DECLARE2 STOP ]
IF :ACTION = [I1] [ INQUIRY1 STOP ]
IF :ACTION = [I2] [ INQUIRY2 STOP ]
IF :ACTION = [I3] [ INQUIRY3 STOP ]
IF :ACTION = [R1] [ REQUEST1 STOP ]
IF :ACTION = [Q1] [ KWIT ]
PRINT [I DON'T UNDERSTAND]
END

TO DECLARE1
MAKE "OBJECT MAKE.NAME :OBJECT
IFELSE NAME? :OBJECT [ MAKE :OBJECT LPUT
  :CLASS THING :OBJECT ]
  [ MAKE :OBJECT LPUT :CLASS [[]] ]
PRINT [-UNDERSTOOD]
END

TO DECLARE2
MAKE "OBJECT MAKE.NAME :OBJECT
IFELSE NAME? OBJECT [ SUB.DECLARE2]
  [ MAKE :OBJECT FPUT (FPUT :PROPERTY
  []) [] ]
MAKE "PROPERTY MAKE.NAME :PROPERTY
MAKE "NAME (WORD :OBJECT ". :PROPERTY)
IFELSE NAME? :NAME [ MAKE :NAME LPUT
  :VALUE THING :NAME ]
  [ MAKE :NAME LPUT :VALUE [] ]
PRINT [-UNDERSTOOD]
END

TO SUB.DECLARE2
IF NOT MEMBER? :PROPERTY FIRST THING
  :OBJECT
  [ MAKE :OBJECT FPUT (LPUT :PROPERTY
  FIRST THING :OBJECT)
  BUTFIRST THING :OBJECT]
END
```

```
TO INQUIRY1
PRINT [-THINKING]
IF NOT NAME? MAKE.NAME :OBJECT
  [ PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP ]
MAKE "DEDUCED "FALSE
DEDUCE1 BUTFIRST THING MAKE.NAME :OBJECT
IF NOT :DEDUCED [ PRINT [-I DO NOT KNOW]
END

TO DEDUCE1 :DEDUCELIST
IF :DEDUCELIST = [] [ STOP ]
IF MEMBER? :CLASS :DEDUCELIST
  [ PRINT [-YES] MAKE "DEDUCED "TRUE
  STOP ]
MAKE "CLASS1 FIRST :DEDUCELIST
IF AND (NOT :CLASS1 = :OBJECT) (NAME?
  MAKE.NAME :CLASS1)
  [ MAKE "DEDUCELIST
  SENTENCE :DEDUCELIST BUTFIRST THING
  MAKE.NAME :CLASS1
DEDUCE1 BUTFIRST :DEDUCELIST
END

TO INQUIRY2
PRINT [-THINKING]
IF NOT NAME? MAKE.NAME :OBJECT
  [ PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP ]
MAKE "NAME (WORD MAKE.NAME :OBJECT ".
  MAKE.NAME :PROPERTY)
IF NAME? :NAME [ IFELSE MEMBER? :VALUE
  THING :NAME
  [ PRINT [-YES] STOP ] [ PRINT [-NO]
  STOP ] ]
MAKE "DEDUCED "FALSE
DEDUCE2 BUTFIRST THING MAKE.NAME :OBJECT
IF NOT :DEDUCED [ PRINT [-I DO NOT KNOW]]
END

TO DEDUCE2 :DEDUCELIST
IF :DEDUCELIST = [] [ STOP ]
MAKE "CLASS1 FIRST :DEDUCELIST
IF AND ( NOT :CLASS1 = :OBJECT) (NAME?
  MAKE.NAME :CLASS1)
  [SUB.DEDUCE2]
DEDUCE2 BUTFIRST :DEDUCELIST
END

TO SUB.DEDUCE2
MAKE "NAME (WORD MAKE.NAME :CLASS1 ".
  MAKE.NAME :PROPERTY)
IF NAME? :NAME [ IFELSE MEMBER? :VALUE
  THING :NAME
  [ PRINT [-YES] ] [ PRINT [-NO] ] MAKE
  "DEDUCED "TRUE STOP ]
MAKE "DEDUCELIST SENTENCE :DEDUCELIST
  BUTFIRST THING MAKE.NAME :CLASS1
END

TO INQUIRY3
PRINT [-THINKING]
IF NOT NAME? MAKE.NAME :OBJECT
  [ PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP ]
MAKE "NAME (WORD MAKE.NAME :OBJECT ".
  MAKE.NAME :PROPERTY)
IF NAME? :NAME [ SUB.INQUIRY3 STOP ]
MAKE "DEDUCE "FALSE
DEDUCE3 BUTFIRST THING MAKE.NAME :OBJECT
IF NOT :DEDUCED [ PRINT [' I DO NOT
  KNOW'] ]
END

TO SUB.INQUIRY3
PRINT (SENTENCE "THE :PROPERTY "OF
  :OBJECT "IS )
PRINT.LIST THING :NAME
END

TO DEDUCE3 :DEDUCELIST
IF :DEDUCELIST = [] [ STOP ]
MAKE "CLASS1 FIRST :DEDUCELIST
IF NOT (AND (NOT :CLASS1 = :OBJECT)
  (NAME? MAKE.NAME :CLASS1))
  [ DEDUCE3 BUTFIRST :DEDUCELIST STOP ]
MAKE "NAME (WORD MAKE.NAME :CLASS1 ".
  MAKE.NAME :PROPERTY)
IF NAME? :NAME [ SUB.DEDUCE3 STOP ]
MAKE "DEDUCELIST SENTENCE :DEDUCELIST
  BUTFIRST THING MAKE.NAME :CLASS1
DEDUCE3 BUTFIRST :DEDUCELIST
END

TO SUB.DEDUCE3
PRINT (SENTENCE "THE :PROPERTY "OF
  :OBJECT "IS )
PRINT.LIST THING :NAME
MAKE "DEDUCED "TRUE
END
```

```
TO REQUEST1
MAKE "OBJECT MAKE.NAME :OBJECT
IF NOT NAME? :OBJECT
  [ PRINT SENTENCE [I KNOW NOTHING
  ABOUT] :OBJECT STOP ]
PRINT SENTENCE :OBJECT "IS
PRINT.LIST BUTFIRST THING :OBJECT
PRINT.PROPS FIRST THING :OBJECT
END

TO PRINT.LIST :LIST
IF :LIST = [] [ STOP ]
(PRINT CHAR 32 CHAR 32 FIRST :LIST)
PRINT.LIST BUTFIRST :LIST
END

TO PRINT.PROPS :LIST
IF :LIST = [] [ STOP ]
MAKE "PROPERTY MAKE.NAME FIRST :LIST
MAKE "NAME (WORD :OBJECT ". :PROPERTY)
PRINT (SENTENCE :OBJECT FIRST :LIST "IS )
PRINT.LIST THING :NAME
PRINT.PROPS BUTFIRST :LIST
END

TO KWIT
REPEAT 4 [PRINT []]
PRINT [-GOODBYE]
REPEAT 4 [PRINT []]
STOPALL
END

TO MAKE.NAME :LIST
IF :LIST = [] [ OUTPUT [] ]
IF (COUNT :LIST) = 1 [ OUTPUT FIRST :LIST
  ]
OUTPUT (WORD FIRST :LIST ". MAKE.NAME
  BUTFIRST :LIST)
END
```

SUPER.INFER
THE PROGRAM
Terrapin Logo PLUS Version

```
TO SUPER.INFER
  INITIALIZE
  INFER.DRIVER
END

TO INITIALIZE
  CLEARTEXT
  PRINT []
  PRINT [WELCOME TO SUPER.INFER]
  PRINT []
  MAKE "QUANTIFIERS [A AN THE MY YOUR ALL
   ONE]
END

TO INFER.DRIVER
  GET.INFO
  CHECK.FORM
  ACT
  INFER.DRIVER
END

TO GET.INFO
  PRINT [WHAT IS OBJECT]
  MAKE "OBJECT REQUEST
  PRINT [WHAT IS CLASS]
  MAKE "CLASS REQUEST
  PRINT [WHAT IS PROPERTY]
  MAKE "PROPERTY REQUEST
  PRINT [WHAT IS VALUE]
  MAKE "VALUE REQUEST
  PRINT [WHAT IS ACTION]
  MAKE "ACTION REQUEST
END

TO CHECK.FORM
  IF NOT LIST? :OBJECT THEN MAKE "OBJECT
   SENTENCE :OBJECT []
  IF NOT LIST? :CLASS THEN MAKE "CLASS
   SENTENCE :CLASS []
  IF NOT LIST? :PROPERTY THEN MAKE
   "PROPERTY SENTENCE :PROPERTY []
  IF NOT LIST? :VALUE THEN MAKE "VALUE
   SENTENCE :VALUE []
  MAKE "OBJECT REMOVE.QUANT :OBJECT
  MAKE "CLASS REMOVE.QUANT .CLASS
  MAKE "PROPERTY REMOVE.QUANT :PROPERTY
  MAKE "VALUE REMOVE.QUANT :VALUE
END
```

```
TO REMOVE.QUANT :LIST
 IF :LIST = [] THEN OUTPUT :LIST
 IF MEMBER? FIRST :LIST :QUANTIFIERS THEN
  OUTPUT BUTFIRST :LIST
 OUTPUT :LIST
END

TO ACT
 IF :ACTION = [D1] THEN DECLARE1 STOP
 IF :ACTION = [D2] THEN DECLARE2 STOP
 IF :ACTION = [I1] THEN INQUIRY1 STOP
 IF :ACTION = [I2] THEN INQUIRY2 STOP
 IF :ACTION = [I3] THEN INQUIRY3 STOP
 IF :ACTION = [R1] THEN REQUEST1 STOP
 IF :ACTION = [Q1] THEN KWIT
 PRINT [I DON'T UNDERSTAND]
END

TO DECLARE1
 MAKE "OBJECT MAKE.NAME :OBJECT
 TEST THING? :OBJECT
 IFFALSE MAKE :OBJECT LPUT :CLASS [[]]
 IFTRUE MAKE :OBJECT LPUT :CLASS THING
  :OBJECT
 PRINT [' UNDERSTOOD']
END

TO DECLARE2
 MAKE "OBJECT MAKE.NAME :OBJECT
 TEST THING? :OBJECT
 IFFALSE MAKE :OBJECT FPUT (FPUT
  :PROPERTY []) []
 IFTRUE IF NOT MEMBER? :PROPERTY FIRST
  THING :OBJECT THEN
 MAKE :OBJECT FPUT (LPUT :PROPERTY FIRST
  THING :OBJECT)
 BUTFIRST THING :OBJECT
 MAKE "PROPERTY MAKE.NAME :PROPERTY
 MAKE "NAME (WORD :OBJECT ". :PROPERTY)
 TEST THING? :NAME
 IFFALSE MAKE :NAME LPUT :VALUE []
 IFTRUE MAKE :NAME LPUT :VALUE THING
  :NAME
 PRINT [' UNDERSTOOD']
END

TO INQUIRY1
 PRINT [' THINKING']
 IF NOT THING? MAKE.NAME :OBJECT
 THEN PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP
 MAKE "DEDUCED "FALSE
 DEDUCE1 BUTFIRST THING MAKE.NAME :OBJECT
 IF NOT :DEDUCED THEN PRINT [' I DO NOT
  KNOW']
END

TO DEDUCE1 :DEDUCELIST
 IF :DEDUCELIST = [] THEN STOP
 IF MEMBER? :CLASS :DEDUCELIST
 THEN PRINT [' YES'] MAKE "DEDUCED "TRUE
  STOP
 MAKE "CLASS1 FIRST :DEDUCELIST
 TEST ALLOF (NOT :CLASS1 = :OBJECT)
  THING? MAKE.NAME :CLASS1
 IFTRUE MAKE "DEDUCELIST
 SENTENCE :DEDUCELIST BUTFIRST THING
  MAKE.NAME :CLASS1
 DEDUCE1 BUTFIRST :DEDUCELIST
END

TO INQUIRY2
 PRINT [' THINKING']
 IF NOT THING? MAKE.NAME :OBJECT
 THEN PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP
 MAKE "NAME (WORD MAKE.NAME :OBJECT ".
  MAKE.NAME :PROPERTY)
 TEST THING? :NAME
 IFTRUE IF MEMBER? :VALUE THING :NAME
  THEN PRINT [' YES'] STOP
  ELSE PRINT [' NO'] STOP
 MAKE "DEDUCED "FALSE
 DEDUCE2 BUTFIRST THING MAKE.NAME :OBJECT
 IF NOT :DEDUCED THEN PRINT [' I DO NOT
  KNOW']
END

TO DEDUCE2 :DEDUCELIST
 IF :DEDUCELIST = [] THEN STOP
 MAKE "CLASS1 FIRST :DEDUCELIST
 TEST ALLOF ( NOT :CLASS1 = :OBJECT)
  THING? MAKE.NAME :CLASS1
 IFTRUE MAKE "NAME (WORD MAKE.NAME
  :CLASS1 ". MAKE.NAME :PROPERTY)
 IFTRUE IF THING? :NAME IF MEMBER? :VALUE
  THING :NAME
 THEN PRINT [' YES'] MAKE "DEDUCED "TRUE
  STOP
 ELSE PRINT [' NO'] MAKE "DEDUCED "TRUE
  STOP
 IFTRUE MAKE "DEDUCELIST
 SENTENCE :DEDUCELIST BUTFIRST THING
  MAKE.NAME :CLASS1
 DEDUCE2 BUTFIRST :DEDUCELIST
END
```

```
TO INQUIRY3
 PRINT [' THINKING']
 IF NOT THING? MAKE.NAME :OBJECT
  THEN PRINT SENTENCE [I DON'T KNOW
  ANYTHING ABOUT] :OBJECT STOP
 MAKE "NAME (WORD MAKE.NAME :OBJECT ".
  MAKE.NAME :PROPERTY)
 TEST THING? :NAME
 IFTRUE PRINT (SENTENCE "THE :PROPERTY
  "OF :OBJECT "IS )
 IFTRUE PRINT.LIST THING :NAME
 IFTRUE STOP
 MAKE "DEDUCE "FALSE
 DEDUCE3 BUTFIRST THING MAKE.NAME :OBJECT
 IF NOT :DEDUCED THEN PRINT [' I DO NOT
  KNOW']
END

TO DEDUCE3 :DEDUCELIST
 IF :DEDUCELIST = [] THEN STOP
 MAKE "CLASS1 FIRST :DEDUCELIST
 TEST ALLOF (NOT :CLASS1 = :OBJECT)
  THING? MAKE.NAME :CLASS1
 IFFALSE DEDUCE3 BUTFIRST :DEDUCELIST
 IFFALSE STOP
 MAKE "NAME (WORD MAKE.NAME :CLASS1 ".
  MAKE.NAME :PROPERTY)
 TEST THING? :NAME
 IFTRUE PRINT (SENTENCE "THE :PROPERTY
  "OF :OBJECT "IS )
 IFTRUE PRINT.LIST THING :NAME
 IFTRUE MAKE "DEDUCED "TRUE
 IFTRUE STOP
 MAKE "DEDUCELIST SENTENCE :DEDUCELIST
  BUTFIRST THING MAKE.NAME :CLASS1
 DEDUCE3 BUTFIRST :DEDUCELIST
END

TO REQUEST1
 MAKE "OBJECT MAKE.NAME :OBJECT
 IF NOT THING? :OBJECT
 THEN PRINT SENTENCE [I KNOW NOTHING
  ABOUT] :OBJECT STOP
 PRINT SENTENCE :OBJECT "IS
 PRINT.LIST BUTFIRST THING :OBJECT
 PRINT.PROPS FIRST THING :OBJECT
END

TO PRINT.LIST :LIST
 IF :LIST = [] THEN STOP
 PRINT SENTENCE [' '] FIRST :LIST
 PRINT.LIST BUTFIRST :LIST
END

TO PRINT.PROPS :LIST
 IF :LIST = [] THEN STOP
 MAKE "PROPERTY MAKE.NAME FIRST :LIST
 MAKE "NAME (WORD :OBJECT ". :PROPERTY)
 PRINT (SENTENCE:OBJECT FIRST :LIST "IS )
 PRINT LIST THING :NAME
 [illegible] TFIRST :LIST
END

TO [illegible]
 REPEAT [PRINT []]
 PRINT [' GOODBYE']
 REPEAT 4 [PRINT []]
 TOPLEVEL
END

TO MAKE.NAME :LIST
 IF :LIST = [] THEN OUTPUT [] STOP
 IF COUNT :LIST = 1 THEN OUTPUT FIRST
  :LIST STOP
 OUTPUT (WORD FIRST :LIST ". MAKE.NAME
  BUTFIRST :LIST)
END
```

8 SUPER.INFER2— A NATURAL LANGUAGE INTERFACE FOR SUPER.INFER

The computer industry is constantly looking to make their machines easy for people to use. One aspect of this is creating an environment in which humans can easily communicate with computers. A solution is to program computers to understand human languages. There are, however, some issues that must be dealt with before a functional program can be developed.

One of these issues is the enormous vocabulary of the English language. There are more words in English than in most other human languages. In order for a computer to accurately parse all sentences, it would at least have to be familiar with all those words. Most computer programs that seem to understand English recognize only a small subset of the vocabulary.

Even if the computer were to be familiar with every word in the English language, there would remain other problems. One of these is ambiguity. Consider the sentence "String flies like arrows." This sentence could have several different meanings. Perhaps something has shot string across the room and you describe it as flying through the air like arrows do. Maybe there is an insect called a "string fly" and maybe string flies have an affection for arrows. And perhaps you work for a sporting goods store and the boss tells you to tie some flies and to make them look like arrows. There are too many meanings and too many ways that words can be arranged for any current computer to accurately interpret every sentence.

A similar problem with understanding human language is the many ideas associated with each word. Even if a computer could be familiar with every meaning and idea connected with every word, and were programmed for every possible sentence structure, the computer would probably not be practical to use. Searching through all those structures, connections, and meanings would take an enormous amount of processing time. You may remember how slow TALK.AND.DRAW was.

Fast machines with larger memories, however, still will not be the solution to the problem of developing computers that can carry on conversations with people. Conversing is a very complex task. In his book *Information Anxiety*, Richard Saul Wurman (1989) discusses the complexities of conversation:

> The implicit and explicit goal of all conversations is understanding. Whether they occur between lovers, friends, relatives, or business associates, conversations have as their express goal to get one's point across, to make a connection between one's thoughts and another person, i.e., the outside world; they are an understanding model, a forum for the exchange of information.
>
> Within conversations are a myriad of self-adjusting systems. As we speak with another person, we constantly readjust our language based on the cues we get from the listener. Do they look baffled or excited, bored or angry?
>
> Unlike almost all machinery, conversations can tune themselves. We make adjustments, simplify, repeat and move between various levels of complexity based on continuous feedback—a quarter-inch nod of the chin, the lowering or raising of eyes, strange guttural noises that say "uh-

huh, hu-huh," blinks, shrugs, turns of the head, loss of eye contact, the making of eye contact. A symphony of signals occurs during even the briefest of conversations.

Until methods for computers to handle the complexities of conversation are developed, programs that operate in human language will be severely limited. Such programs will rely on tricks to "understand" human language. One trick is to search the sentence for some key word or words that would clue the computer into executing the appropriate action. DOCTOR and TALK.AND.DRAW were programs that used this key-word analysis.

Another trick is to create a microworld. It is the enormity of the English language that limits natural language processing, but if the language were used just for some limited application, then the relative vocabulary and applicable grammar would be considerably smaller. BLOCKS and SHRDLU use this short-cut.

Most commercial software uses a combination of techniques. Many of the more powerful natural language processors break down sentences to understand them. This process is called parsing and is very similar to the diagraming of sentences that students used to do in the middle grades. In this process, the computer must first break up a sentence before it can attempt to analyze the meaning. Let's look at how a computer might break up a sentence. Suppose the sentence were:

THE LONG THIN DOG SAT
BEHIND THE WHITE HOUSE

The parser would do two things: break up the sentence into its parts, and indicate the parts that are bound together. Some systems would do this using brackets as shown:

```
[ [THE [ [LONG] [ [THIN] [DOG] ] ] ] SAT
[ [BEHIND] [THE [ [WHITE] [HOUSE] ] ] ] ]
```

This appears confusing at first, but by grouping bound parts within brackets, the machine can begin to understand the meaning of the sentence. For example, WHITE is bound to HOUSE because it modifies the house. In the noun phrase, LONG and THIN are bound to DOG as modifiers. The subject of the sentence is the entire noun phrase [[THE [[LONG] [[THIN] [DOG]]]]. This is bound to the right-hand side with the verb SAT. This can all be symbolized in the structure of a tree.

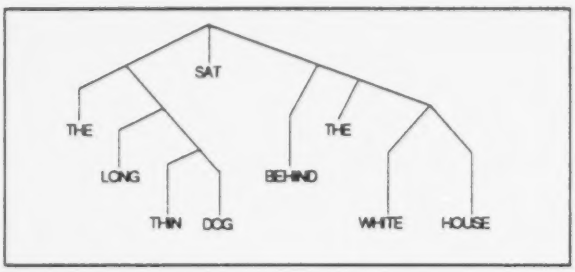


DIAGRAM 1

Regardless of how a system represents sentence structures, it requires rules for breaking up the sentences. These rules are really considered the grammar. Below is a grammar for a subset of English.

A GRAMMAR FOR A SUBSET OF ENGLISH

```
SENTENCE          → NOUN.PHRASE
                    VERB.PHRASE
NOUN.PHRASE       → QUANTIFIER  ADJ.NOUN.
                    PHRASE
NOUN.PHRASE       → ADJ.NOUN.PHRASE
VERB.PHRASE       → Verb  NOUN.PHRASE
                    ADVERB.  PHRASE
                    PREP.PHRASE
VERB.PHRASE       → Verb  ADVERB.PHRASE
                    PREP.PHRASE
VERB.PHRASE       → Verb
QUANTIFIER        → Article
QUANTIFIER        → Possessive
ADJ.NOUN.PHRASE   → Adjective
                    ADJ.NOUN.PHRASE
ADJ.NOUN.PHRASE   → NOUN
ADVERB.PHRASE     → Adjective
                    ADVERB.PHRASE
ADVERB.PHRASE     → Adverb
PREP.PHRASE       → Preposition
                    NOUN.PHRASE
NOUN              → Regular.Noun
NOUN              → Proper.Noun
NOUN              → Pronoun
```

The rules in all capitals are further defined later in the list. For example, a NOUN can be either a regular noun, a proper noun, or a pronoun. Each of these three sub-rules were not written in all capitals. That is because these are terminal rules; that is, these do not require further definition. A regular noun is any noun, such as "boat" or "car" or "dog." A proper noun would be any name, such as "Sears Tower" or "Sally" or "Ireland." These terminal rules are the connection between the grammar and the vocabulary.

A program designed to accept this grammar can understand sentences like these: Sally eats fish quietly at the table. Regrettable children cry. Pickled pig's feet smell like dead fish. Eager employees vie for big raises.

When a grammar is created for a natural language interface it must be designed with its application in mind. This can help minimize the problems discussed earlier that face natural language processors. When only the types of sentences absolutely needed for a program are considered, the programmer can greatly reduce the total size of the grammar. There are usually fewer rules to consider, and the applicable vocabulary is smaller. This is similar to the microworld concept we used in BLOCKS.

The grammar designed for SUPER.INFER was created in this way. There are only four types of actions, so each sentence is redefined four ways (see below). There are two types of declarations, so DECLARATION is redefined only twice. Likewise, INQUIRY has three definitions. Here is the complete grammar.

A GRAMMAR FOR SUPER.INFER

```
SENTENCE          → DECLARATION
SENTENCE          → INQUIRY
SENTENCE          → REQUEST
SENTENCE          → Quit
DECLARA.ION       → OBJECT  Is  CLASS
DECLARATION       → OBJECT's  Property  Is
                    Value
INQUIRY           → What  is  OBJECT's
                    Property
INQUIRY           → Is  OBJECT's  Property
                    Value
INQUIRY           → Is  OBJECT  CLASS
REQUEST           → Tell  me  about  OBJECT
OBJECT            → Thing
OBJECT            → CLASS
CLASS             → Quantifier  Thing
```

The italicized letters and words are specific strings of characters that make up parts of some types of sentences. These are called literals because they must appear in the sentence as they are literally defined in the grammar. In fact, our program uses these literals to determine which kind of action to take and to extract the various parts from each sentence.

SUPER.INFER still creates the links between classes and properties as well as performing the deduction for various inquiries. The new grammar, when implemented in Logo, will allow the user to type English sentences instead of manually entering the OBJECT, CLASS, and so forth. This grammar will, for instance, interpret sentences such as the following: Tell me about big dogs. Is Rover's color black and white? Spot is a cocker spaniel. A lab's color is black. Is Rover a mammal? What is Spot's color?

How the Program Works

This program requires all of the procedures from the program in the last chapter. The procedures discussed and listed below are meant to be added to the others. You might want to call this revised program SUPER.INFER2 so that you can keep the two versions separate on your data disk.

Once you call up SUPER.INFER and save a copy as SUPER.INFER2, delete the old GET.INFO, which asked the questions of the user. The new GET.INFO will take sentences for input and parse them to find :ACTION, :OBJECT, :CLASS, :PROPERTY, and :VALUE. First, GET.INFO initializes those five important variables to the empty list using INIT. Next, GET.SENTENCE is called to get the sentence from the user.

Now comes the language analysis. The program can get hints as to sentence type by looking at the first word in the sentence. Below is a chart that shows what assumptions are made based on the first word in the sentence.

If the sentence is "Quit," then the user wants to stop the conversation with the computer. If the first word is "tell," then the sentence is likely to be a request for information on an object. If the word is "what," then the user intended to use the third type of inquiry. When the first word is "is," then the sentence is one of the other two types of inquiries. If the sentence doesn't match any of these patterns, then the program assumes that the sentence is a declaration.

```
TO GET.INFO
INIT
GET.SENTENCE
IF (FIRST :INPUT) = "WHAT [I3.PARSE STOP]
IF (FIRST :INPUT) = "TELL [R1.PARSE STOP]
IF (FIRST :INPUT) = "IS [I.PARSE STOP]
IF (FIRST :INPUT) = "QUIT [MAKE "ACTION
  [Q1] STOP]
D.PARSE
END

TO INIT
MAKE "OBJECT []
MAKE "CLASS []
MAKE "PROPERTY []
MAKE "VALUE []
MAKE "ACTION []
END
```

FIRST WORD IN SENTENCE	TYPE OF ACTION	PROCEDURE CALLED
What	third inquiry	E3.PARSE
Tell	request for information	R1.PARSE
Is	first or second inquiry	E.PARSE
Quit	quit the program	none
anything else	declaration	D.PARSE

```
TO  GET.SENTENCE
PRINT  []
PRINT  []
PRINT  ["=>]
MAKE  "INPUT  READLIST
PRINT  []
END
```

If the first word is "What," then E3.PARSE is called to extract the values for :ACTION, :OBJECT, and :PROPERTY. For example, assume that "What is Spot's color" is the user's input. :ACTION is made [E3]. The first two words in the sentence are then removed since they should be "What is." :INPUT is now [Spot's color]. :OBJECT is extracted and the "'s" removed by the procedure GET.OBJECT leaving :OBJECT as [Spot] and :INPUT as [color]. As you can see in our grammar above, the property is then made up of the remaining words.

GET.OBJECT is used by many of the other parsing procedures to extract :OBJECT. This procedure recognizes several ways of telling where in the sentence the object ends. Look to the grammar above and you will see that sometimes the end of the object is signaled by the end of the sentence, sometimes by the word "is," and sometimes by a quantifier. In several cases, the object ends with the possessive "'s." This was tricky to code into the program since Logo is finicky about how it handles an apostrophe. The ASCII code equivalent had to be called in.

```
TO  I3.PARSE
MAKE  "ACTION  [I3]
MAKE  "INPUT  BUTFIRST  BUTFIRST  :INPUT
GET.OBJECT
MAKE  "PROPERTY  :INPUT
END

TO  GET.OBJECT
IF  EMPTY?  :INPUT  [STOP]
MAKE  "X  FIRST  :INPUT
IF  :X = "IS  [STOP]
IF  AND  (MEMBER?  :X  :QUANTIFIERS)  (NOT
   EMPTY?  :OBJECT)  [STOP]
MAKE  "INPUT  BUTFIRST  :INPUT
IF  (COUNT  :X)  >2  [IF  (WORD  LAST  BUTLAST
   :X  LAST  :X)  =  (WORD  CHAR  39  "S )
   [MAKE  "OBJECT  SENTENCE  :OBJECT  BUTLAST
   BUTLAST  :X  STOP]]
MAKE  "OBJECT  SENTENCE  :OBJECT  :X
GET.OBJECT
END

TO  R1.PARSE
MAKE  "ACTION  [R1]
MAKE  "INPUT  BUTFIRST  BUTFIRST  BUTFIRST
   :INPUT
MAKE  "OBJECT  :INPUT
END
```

When the first word in the user's sentence is "tell," the computer assumes that the user has typed a sentence beginning with the phrase "Tell me about..." This is a request for all the information on an object. If the user were to enter "Tell me about big dogs," R1.PARSE would make :ACTION [R1]. The first three words are stripped away and the remaining word or words are put into :OBJECT. Here, :OBJECT would be [big dogs].

The sentences beginning with the word "is" require a little more analysis. They are inquiries, but some are looking for a class (INQUIRY1) or for a property value (INQUIRY2). In the case of the sentence "Is Rover a mammal," because "Is" is the first word, E.PARSE is called and the first word is removed. Both types of inquiry start with the object, so GET.OBJECT is called. :OBJECT is now [Rover] and :INPUT [a mammal]. If the next word is a quantifier, as it is here, then the sentence is recognized as looking for a class. :ACTION is made [E1], and :CLASS is made the remaining part of the sentence, less the article. :CLASS would be [mammal].

Consider the inquiry "Is Rover's color black and white?" I.PARSE removes "Is" and GET.OBJECT makes :OBJECT [Rover]. The next word, however is not a quantifier. :ACTION is made [E2] and :INPUT remains [color black and white]. Then comes a

small problem with the program. There are two pieces of information remaining in the sentence: the property and the property value. There is no easy way for the computer to determine where the property ends and the value begins. As a crutch, the program assumes that the user only uses a single-word property. In our example :PROPERTY becomes [color]. Everything after that word is then put into :VALUE. This is an easy, although limiting, way of dealing with the problem of telling the property from the value. Another solution is discussed in the Extensions section below.

```
TO I.PARSE
MAKE "INPUT BUTFIRST :INPUT
GET.OBJECT
IF MEMBER? FIRST :INPUT :QUANTIFIERS
   [MAKE "ACTION [I1]
   MAKE "CLASS :INPUT STOP]
MAKE "ACTION [I2]
MAKE "PROPERTY FIRST :INPUT
MAKE "VALUE BUTFIRST :INPUT
END

TO D.PARSE
GET.OBJECT
IF (FIRST :INPUT) = "IS [MAKE "ACTION
   [D1] MAKE "CLASS BUTFIRST
   :INPUT STOP]
MAKE "ACTION [D2]
GET.PROP
MAKE "VALUE :INPUT
END

TO GET.PROP
IF EMPTY? :INPUT [STOP]
MAKE "X FIRST :INPUT
MAKE "INPUT BUTFIRST :INPUT
IF :X = "IS [STOP]
MAKE "PROPERTY SENTENCE :PROPERTY :X
GET.PROP
END
```

D.PARSE behaves something like I.PARSE, only D.PARSE analyses declaratives instead of inquiries. Given that the user's sentence is "Spot is a Cocker Spaniel," D.PARSE begins by getting the object: [Spot]. Next it looks for the word "is." If it is there, as in our example, :ACTION becomes [D1] and :CLASS becomes the rest of the sentence after "is" and any quantifiers are removed: [Cocker Spaniel].

If the sentence were "A lab's color is black," GET.OBJECT would make :OBJECT [lab]. The word "is" is not next so :ACTION becomes [D2] and GET.PROP is called. At this point :INPUT is [color is black]. GET.PROP looks through the sentence looking for the word "is." In this type of sentence, "is" separates the property from the property value. :PROPERTY becomes [color]. GET.PROP does not assume that the property will be a single word. It was designed this way partly so that if you improve the parsing for INQUIRY2, DECLARE2 would already be set up for multiword properties. A one-word property name is not needed for a crutch since the word "is" is a clear clue as to where the property ends and the value begins. After "is" is removed, the remaining part of the sentence is placed in :VALUE : [black].

Possible Extentions and Expansions

- Teach the computer to handle punctuation. Right now, neither the computer nor the user uses commas, question marks, or periods. One way to effect the change is to have the computer go through the user's sentence and remove any punctuation before parsing.
- Improve the computer's responses to be more like sentences. This will make the interaction feel more like that between two intelligent beings.
- Teach the computer to deal with pronouns. Perhaps one sentence is "The big dog is a vicious animal" and the next sentence is "He is a rabid beast." "He" would refer to "big dog." The computer could make this distinction by keeping

track of whatever the last object was.

- Improve the parsing so that the system doesn't crash. If it gets a pattern it doesn't recognize, make sure that the computer responds with "I don't understand" or some such message. There is almost no error trapping now. Try to be as thorough as you can since it can make the computer seem more intelligent.
- Take care of the "one-word property problem." One possibility is to keep a list of all the various properties and, while parsing the inquiry, take one word at a time until it matches one of the properties in the list.
- Have the computer understand plural as well as singular nouns. Part of the solution would be to remove the plural "s" on the end of a plural noun. Another part is to recognize "are" to mean the same as "is." Be careful; you might have to look out for special cases.
- If you built in other types of links for SUPER.INFER, design and implement the appropriate grammar for them.

SUPER.INFER2
THE ADDITIONAL PROCEDURES
LogoWriter Version

```
TO GET.INFO
INIT
GET.SENTENCE
IF (FIRST :INPUT) = "WHAT [I3.PARSE STOP]
IF (FIRST :INPUT) = "TELL [R1.PARSE STOP]
IF (FIRST :INPUT) = "IS [I.PARSE STOP]
IF (FIRST :INPUT) = "QUIT [MAKE "ACTION
  [Q1] STOP]
D.PARSE
END

TO INIT
MAKE "OBJECT []
MAKE "CLASS []
MAKE "PROPERTY []
MAKE "VALUE []
MAKE "ACTION []
END

TO GET.SENTENCE
PRINT []
PRINT []
PRINT ["=>]
MAKE "INPUT READLIST
PRINT []
END

TO I3.PARSE
MAKE "ACTION [I3]
MAKE "INPUT BUTFIRST BUTFIRST :INPUT
GET.OBJECT
MAKE "PROPERTY :INPUT
END

TO GET.OBJECT
IF EMPTY? :INPUT [STOP]
MAKE "X FIRST :INPUT
IF :X = "IS [STOP]
IF AND (MEMBER? :X :QUANTIFIERS) (NOT
  EMPTY? :OBJECT) [STOP]
MAKE "INPUT BUTFIRST :INPUT
IF (COUNT :X) >2 [IF (WORD LAST BUTLAST
  :X LAST :X) = (WORD CHAR 39 "S )
  [MAKE "OBJECT SENTENCE :OBJECT BUTLAST
  BUTLAST :X STOP]]
MAKE "OBJECT SENTENCE :OBJECT :X
GET.OBJECT
END
```

```
TO  R1.PARSE
MAKE  "ACTION  [R1]
MAKE  "INPUT  BUTFIRST  BUTFIRST  BUTFIRST
   :INPUT
MAKE  "OBJECT  :INPUT
END

TO  I.PARSE
MAKE  "INPUT  BUTFIRST  :INPUT
GET.OBJECT
IF  MEMBER?  FIRST  :INPUT  :QUANTIFIERS
   [MAKE  "ACTION  [I1]
   MAKE  "CLASS  :INPUT  STOP]
MAKE  "ACTION  [I2]
MAKE  "PROPERTY  FIRST  :INPUT
MAKE  "VALUE  BUTFIRST  :INPUT
END

TO  D.PARSE
GET.OBJECT
IF  (FIRST  :INPUT)  =  "IS  [MAKE  "ACTION
   [D1]
   MAKE  "CLASS  BUTFIRST  :INPUT  STOP]
MAKE  "ACTION  [D2]
GET.PROP
MAKE  "VALUE  :INPUT
END

TO  GET.PROP
IF  EMPTY?  :INPUT  [STOP]
MAKE  "X  FIRST  :INPUT
MAKE  "INPUT  BUTFIRST  :INPUT
IF  :X  =  "IS  [STOP]
MAKE  "PROPERTY  SENTENCE  :PROPERTY  :X
GET.PROP
END
```

SUPER.INFER2
THE ADDITIONAL PROCEDURES
Terrapin Logo PLUS Version

```
TO  GET.INFO
  INIT
  GET.SENTENCE
  IF  FIRST  :INPUT  =  "WHAT  THEN  I3.PARSE
   STOP
  IF  FIRST  :INPUT  =  "TELL  THEN  R1.PARSE
   STOP
  IF  FIRST  :INPUT  =  "IS  THEN  I.PARSE  STOP
  IF  FIRST  :INPUT  =  "QUIT  THEN  MAKE
   "ACTION  [Q1]  STOP
  D.PARSE
END

TO  INIT
  MAKE  "OBJECT  []
  MAKE  "CLASS  []
  MAKE  "PROPERTY  []
  MAKE  "VALUE  []
  MAKE  "ACTION  []
END

TO  GET.SENTENCE
  PRINT  []
  PRINT  []
  PRINT1  "=>
  MAKE  "INPUT  REQUEST
  PRINT  []
END

TO  I3.PARSE
  MAKE  "ACTION  [I3]
  MAKE  "INPUT  BUTFIRST  BUTFIRST  :INPUT
  GET.OBJECT
  MAKE  "PROPERTY  :INPUT
END
```

```
TO  GET.OBJECT
  IF  EMPTY?  :INPUT  THEN  STOP
  MAKE  "X  FIRST  :INPUT
  IF  :X  =  "IS  THEN  STOP
  IF  ALLOF  MEMBER?  :X  :QUANTIFIERS  NOT
   EMPTY?  :OBJECT  THEN  STOP
  MAKE  "INPUT  BUTFIRST  :INPUT
  IF  COUNT  :X  >2  IF  WORD  LAST  BUTLAST  :X
   LAST  :X  =  WORD  CHAR  39  "S
   THEN  MAKE  "OBJECT  SENTENCE  :OBJECT
   BUTLAST  BUTLAST  :X  STOP
  MAKE  "OBJECT  SENTENCE  :OBJECT  :X
  GET.OBJECT
END

TO  R1.PARSE
  MAKE  "ACTION  [R1]
  MAKE  "INPUT  BUTFIRST  BUTFIRST  BUTFIRST
   :INPUT
  MAKE  "OBJECT  :INPUT
END

TO  I.PARSE
  MAKE  "INPUT  BUTFIRST  :INPUT
  GET.OBJECT
  IF  MEMBER?  FIRST  :INPUT  :QUANTIFIERS
   THEN  MAKE  "ACTION  [I1]  MAKE  "CLASS
   :INPUT  STOP
  MAKE  "ACTION  [I2]
  MAKE  "PROPERTY  FIRST  :INPUT
  MAKE  "VALUE  BUTFIRST  :INPUT
END

TO  D.PARSE
   GET.OBJECT
  IF  FIRST  :INPUT  =  "IS  THEN  MAKE  "ACTION
   [D1]  MAKE  "CLASS  BUTFIRST  :INPUT  STOP
  MAKE  "ACTION  [D2]
  GET.PROP
  MAKE  "VALUE  :INPUT
END

TO  GET.PROP
  IF  EMPTY?  :INPUT  THEN  STOP
  MAKE  "X  FIRST  :INPUT
  MAKE  "INPUT  BUTFIRST  :INPUT
  IF  :X  =  "IS  THEN  STOP
  MAKE  "PROPERTY  SENTENCE  :PROPERTY  :X
  GET.PROP
END
```

9 VEXPERT— AND EXPERT SYSTEMS

If programs that move blocks, discuss your problems, or tell you if ostriches can fly don't seem very practical, don't worry. This chapter is going to focus on one of the most practical areas of artificial intelligence—expert systems. Like human experts, expert systems can make diagnoses, recommend actions, and draw conclusions. This field of AI is so well developed that professionals in business, industry, law, and medicine already rely on expert systems.

One of the main ideas behind expert systems is to try to make accessible, via the computer, the wisdom and knowledge of an expert to someone without the expert's extensive background. To do this the system should have three functions: problem solving, interactive explanation, and learning new knowledge. The expert system can then be used to train people, to advise professionals, and to replace people when the expert system's performance is better than that of human experts.

Imagine a doctor in a small rural hospital being able to enter a patient's symptoms at a terminal and, with the use of an expert system, diagnose a blood disease that only a specialist would normally spot. Imagine a geologist typing in the results of tests for an area and the computer telling the geologist what types of ores are likely to be found in that area. Imagine young students learning subtraction on a computer that can determine an individual student's arithmetic weaknesses and then help the student to overcome them. Imagine a computer salesman who, with the help of an expert system, can configure a complicated computer system in under a minute, a task that would have taken 20-30 minutes without computer assistance.

You actually don't need to imagine very hard. All of these expert systems already exist.

Chemistry has the honor of being the first field for which an expert system was created. The project was started at Stanford in 1965 by a professor of genetics named Joshua Lederberg and a computer expert called Edward A. Feigenbaum. Feigenbaum was interested in the ways "humans construct hypotheses and build models based on data and observations about the world" (Peat, 1988) and had worked on learning systems before coming to Stanford. Lederberg was director of the Kennedy Laboratories of Molecular Medicine at Stanford.

They set out to create a computer program that could determine exact or likely structures of molecules based on chemical formulae and spectroscopic data. Molecules are made up of atoms put together much like building blocks. Atoms follow specific rules for joining with other atoms; carbon, for instance, can only bond with up to four other atoms. Each atom, however, has a different rule for bonding with other atoms, and all the possible combinations (carbon alone has thousands of combinations) make figuring out the makeup of any new compound or molecule very difficult.

Tests must be run to determine what a new compound might be. Not only must scientists test for specific types of atoms and compounds, but they must use experience and educated guesswork to pin down the composition. This is very time consuming. Often, unless the scientists make a lucky guess, they run many tests that run into dead ends before testing for the correct elements. Lederberg and

Feigenbaum created a program they called *DENDRAL* that could determine the composition of a compound using not only the rules about how atoms bond, but tricks expert chemists use, heuristics, and general chemical knowledge. The program was running by 1968 and performed better than any person in the field at the time (Hartnell, 1985).

By 1976, *DENDRAL* was updated by the programs *COGEN* and *REACT*. These expert systems are now in routine use in laboratories worldwide. Other expert systems in the field of chemistry include *CRYSALIS*, used for X-ray crystallography, and *SYNCHEM*, aiding in synthetic chemistry.

Medicine is another field rich in expert systems. The program considered the father of the modern expert systems is *MYCIN*. *MYCIN*, also developed at Stanford, diagnoses blood and meningitis infections and is considered as good as human experts in the field. This is especially useful for a physician who isn't an expert on blood infections. He can use his computer in an interactive session to pin down the cause of the infection and to prescribe treatment.

MYCIN begins by asking the doctor for general information about the patient, including name, gender, and age. *MYCIN* then asks about symptoms, test results, treatment, and other general medical information. Once *MYCIN* makes a diagnosis, it tries to determine the causes of the disease. Finally *MYCIN* suggests treatment. *MYCIN* also has the ability to explain, at any point in a session, why it asked a specific question, how it determined a diagnosis, or why it has prescribed a treatment. Below is a sample dialog with *MYCIN* (Peat, 1988).

MYCIN: Patient's name (first-last):

DOCTOR: Fred Smith

MYCIN: Sex:

DOCTOR: Male

MYCIN: Age:

DOCTOR: 55

MYCIN: Have you been able to obtain positive cultures from a site at which Fred Smith has an infection:

DOCTOR: Yes

MYCIN: What is the infection?

DOCTOR: Primary-bacteremia

MYCIN goes on to ask the doctor some specific questions about the specimen and the tests that were performed on it, and then MYCIN gives its diagnosis.

MYCIN: My therapy recommendation will be based on the following probable infections(s) and potential causative organism(s):

```
INFECTION-1 is PRIMARY BACTEREMIA
<Item 1> The identity of ORGANISM-1
may be PSEUDOMONAS-AERUGINOSA
<Item 2> The identity of ORGANISM-1
may be KLEBSIELLA-PNEUMONIAE
<Item 3> The identity of ORGANISM-1
may be E.COLI
<Item 4> The identity of ORGANISM-1
may be BACTERIODES-FRAGILIS
<Item 5> The identity of ORGANISM-1
may be PROTEUS NON-MIRABILIS
```

On a scale of 0 to 4, where higher numbers indicate increasing severity, how would you rate Fred Smith's degree of sickness?

DOCTOR: 3

MYCIN: Does Fred Smith have a clinically significant allergic reaction to any antimicrobial agent:

DOCTOR: No

MYCIN: Fred Smith's weight in kilograms (or <number> POUNDS):

DOCTOR: 70

MYCIN: (Rec 1) My preferred therapy recommendation is as follows:
In order to cover for items <1 2 3 5 6>:
Give: GENTAMICIN
Dose: 119 mg (1.7 mg/kg) q8h IV (or IM) for 10 days
Comments: Modify dose in renal failure
In order to cover for item <4>:
Give: CLINDAMYCIN
Dose: 595 mg (8.5 mg/kg) q6h IV (or IM) for 14 days
Comments: If diarrhea or other GI symptoms develop, patient should be evaluated for possible PSEUDOMEMBRANOUS COLITIS.
Do you wish to see the next-choice therapy?

DOCTOR: Yes

MYCIN continues to suggest alternative therapies. With some versions of MYCIN, the doctor can continue his conversation and press the system for details.

MYCIN: Is the patient's illness with ORGANISM-1 a hospital-acquired infection?

DOCTOR: Why? (i.e., Why is it important to determine whether the infection with ORGANISM-1 was acquired while the patient was hospitalized?)

MYCIN: This will aid in determining the category of ORGANISM-1. It has already been established that
(1.1) the gram stain of ORGANISM-1 is gramneg, and
(1.2) the morphology of ORGANISM-1 is rod, and
(1.3) the aerobicity of ORGANISM-1 is facultative
Therefore, if
(1.4) the infection with ORGANISM-1 was not acquired while the patient was hospitalized then there is strongly suggestive evidence (.8) that the category of ORGANISM-1 is enterobacteriacae. Also there is weakly suggestive evidence (.1) that the identity of ORGANISM-1 is PSEUDOMONAS-AERUGINOSA (RULE 27)

DOCTOR: How 1.3? (i.e., How was it established that the aerobicity of ORGANISM-1 is facultative?)

MYCIN: The following was used:
(3.1) RULE 27 indicated there is strongly suggestive evidence (.8) that the aerobicity of ORGANISM-1 is facultative.
(3.2) RULE 27 indicated that there is weakly suggestive evidence (.2) that the aerobicity of ORGANISM-1 is anerobic.
Since this gave a cumulative CF (Certainty Factor) of (.8) for facultative, and (.2) for anerobic, it has been established that the aerobicty of ORGANISM-1 is facultative.

This dialog with *MYCIN* illustrates some interesting points about expert systems. Many systems allow the user to ask "how" or "why." Although most systems in common use have been thoroughly tested and are of at least human expert accuracy, people, not machines, should be in control

over important things like the diagnosis of diseases. Being able to ask "how" and "why" lets the physician decide if the computer's "thinking" is reasonable. Even though the computer simply responds with the rule it is using to determine a diagnosis, it lets the user know how it reached its decision.

Another interesting characteristic of many expert systems revealed in the dialog with *MYCIN* is a phenomenon known as "fuzzy reasoning." When people make analyses or diagnoses, they are often based on intuition, opinions, and judgments, as well as logic and "hard" facts. An auto mechanic, for example, knows that if a car's climate control system is not producing any heat, the thermostat is probably defective. Recall that in the conversation with *MYCIN*, it based some decisions on an 80% certainty that the aerobicity of the cause of the infection was facultative.

The dialog also refers to "RULE 27." Expert systems use a combination of a large collection of information from some field (like blood infections) and programmed logic. The collection of information is usually in the form of production rules. For example:

```
IF your car engine stops
AND
IF the gas gauge points to E

THEN you ran out of gas.
```

MYCIN uses nearly 450 production rules.

Early AI researchers had tried to develop programs that could solve any problem. The researchers believed that programs didn't need a lot of data to solve problems and would only need to be programmed with the general rules of reason and logic (not to be confused with the knowledge-specific production rules discussed above). The researchers soon realized that they were wrong; programs, like people, needed access to lots of information to solve problems.

A great deal of the programming of *MYCIN* manipulates the production rules that help *MYCIN* diagnose. The production rules are specific to blood infections, but the programmers at Stanford noticed that the procedures used to manipulate the production rules to draw conclusions were fairly general and could be used on any set of domain-specific production rules. These procedures were the programmed logic part of the program. The programmers modified the program by removing the blood infection rules and called it *EMYCIN* (for Empty *MYCIN*). A computer operator can use *EMYCIN* to enter a new set of rules, creating a new expert system. Any program that lets a person do this is called an expert system shell. *EMYCIN* was one of the first, but there are many in use today. By cutting down on the amount of programming that needs to be done, expert system shells greatly speed up the process of creating an expert system.

PUFF is another medical expert system. It is used for diagnosing pulmonary diseases. *PUFF* has a lot in common with MYCIN; it was developed at Stanford and written using *EMYCIN*. In 1982, *PUFF* was the only medical expert system in regular clinical use for clinical cases (Chabris, 1987). It needs only about 55 rules to make its diagnoses. *PUFF* has produced the same set of diagnoses as a human pulmonary expert in at least one trial of 150 cases (Hartnell, 1985).

There are many other medical expert systems. *Chest Pain* tries to find the cause of chest pain. It was designed by cardiologist Evlin Kinney and is a better diagnostician than emergency room physicians. MIT's *Digitalis Therapy Advisor* prescribes treatment for those using digitalis. The doctor continues to interact with the program, as with a human expert, for the

entire time the patient is undergoing treatment. An expert system used for internal medicine is called *Caduceus* and was developed at the University of Pittsburgh. It maintains over 100,000 rules and is considered an expert at 85% of all internal diseases (Chabris, 1987). *Caduceus* can diagnose, but it does not prescribe treatment.

Guidon is another program closely linked to *MYCIN*. As *EMYCIN* uses the programmed logic of *MYCIN*, *Guidon* uses its production rules. *Guidon* is an educational program, created at Stanford, that is used by medical students to learn the diagnosis and treatment of bacteriological infections. *Guidon* gives a medical student the symptoms of a simulated case. The medical student can then ask questions of the program. The student makes a diagnosis, and the computer evaluates the student's work.

Many teachers are already familiar with CAI or CAL (Computer Assisted Instruction or Computer Assisted Learning). Traditional CAI usually consists primarily of drill and practice programs designed to help students review certain facts or learn basic skills. Those CAI programs simply count how many questions the student answered correctly, although some are also programmed to come up with questions randomly. ICAI, Intelligent Computer Assisted Instruction, uses expert systems to help diagnose students' difficulties and weaknesses and then prescribe specific tutorials and drills.

BUGGY is an early example of ICAI. It was developed at BBN (Bolt, Beranek, and Newman, Cambridge, Massachusetts) in the late 1970s by John Seely Brown, Richard Burton, and Kathy Larkin. It diagnoses students' problems with subtraction and operates on the idea that mistakes occur when students are following the wrong procedures. BUGGY gives the student a series of problems, analyzes answers, and looks for error patterns to determine the cause of the errors.

Another ICAI program was developed for the U.S. Navy by BBN. *Steamer* uses graphics to teach sailors how to operate steam engines. It provides sailors with a visual conception of how steam moves through pipes and what happens when valves are opened or closed. Sailors can even experiment by opening or closing valves during simulated emergencies to see what would happen. The Navy has cut down significantly on the number of steam-related accidents since implementing Steamer (Muir, 1985).

Mathematics has been a realm into which AI researchers have delved for a long time. At the same time that the team at Stanford was working on *DENDRAL*, a team at MIT was working on an expert system to help solve mathematical problems. Carl Engleman, William Martin, and Joel Moses are the creators of *MACSYMA*, the mathematical problem solver. As calculators make quick work of arithmetic, *MACSYMA* makes quick work of solving, simplifying, transforming, and graphing the more complex equations and expressions of trigonometry and calculus.

Programs like *MACSYMA* are called Computer Algebra Systems by mathematicians and Symbolic Manipulators by artificial intelligence researchers. There are several currently in wide use; *DERIVE*, *Maple*, and *Mathematica* are a few. A mathematician who wishes to work with a new equation must first figure out into which category the equation fits. The mathematician must do this by examining the form of the equation and probably the graph. Once the equation has been classified, various operations can be applied. When working on painstaking calculations by hand, the mathematician must be careful not to make any mistakes. A

Computer Algebra System can very quickly classify an equation and, as we know from using calculators, performs mathematical operations speedily and flawlessly.

As with Computer Algebra Systems, many expert systems can perform functions more quickly than human experts. Digital Equipment Corporation (DEC) manufactures minicomputers. When an institution buys a computer system, DEC has to put together a system of components to meet the customer's needs. It takes a DEC technician 20-30 minutes to design an appropriate system. *XCON*, developed jointly by DEC and Carnegie Mellon University, can do the same work in under a minute (Mishkoff, 1988). It also calculates the lengths of needed cables and draws detailed diagrams of the system configuration for the manufacturing team. *XCON* was first built in 1980 and is considered the first expert system used routinely in an industrial setting. *XCON* cost DEC nearly $2 million to develop but saved the company nearly $10 million in four years' use (Peat, 1988).

Tim Hartnell (1985) relates another story of how an expert system quickly paid for itself:

> Feigenbaum and McCorduck in *The Fifth Generation...* give a graphic example of the "earning-back" power of major expert systems. They cite the case of a major American company that had recently bought an expert system designed to diagnose failures in particular types of electricity generating plants. Testing an early, and largely incomplete, version of the program against the real data that led to one of the company's plants being shut down in 1981, it was found the system discovered the cause of the problem that led to the shutdown in a matter of seconds. It had taken the human experts working at that plant days to come to the same conclusion. In the meantime, the plant had been shut down for four days, a closure that cost the company around $1.2 million.

But time saved is not the only way expert systems pay for themselves. *PROSPECTOR* helps geologists with mineral exploration and was created at SRI International in the late 1970s. It has over a dozen models of ore deposit patterns and can help predict where there are likely to be deposits. *PROSPECTOR* is credited with the discovery of large deposits of molybdenum in the Pacific Northwest and Canada. Geologists did not even guess that there was any molybdenum there before using the expert system. The 1982 find has been valued at $100 million (Peat, 1988).

A lot of time and effort goes into developing an expert system before it starts paying for itself. It may take a team over a year to develop a sophisticated system. Much of that time is used to discover and organize the information necessary to solve problems in some specific field. A team is made up of two groups of people: the knowledge engineers and the domain experts. The domain experts are the doctors, geologists, mathematicians, or scientists who are experts within their fields. The knowledge engineers observe and interview the domain experts to discover what information should be included in the expert system and then are responsible for creating the actual program. Here is a sample interview (Chabris, 1987).

> **KNOWLEDGE ENGINEER:** Suppose you were told that a spill had been detected in White Oak Creek one mile before it enters White Oak Lake. What would you do to contain the spill? [It is a good idea to start with a

concrete example in order to observe the expert in simulated action.]

DOMAIN EXPERT: That depends on a number of factors. [A typical response at this point.] I would need to find the source in order to prevent the possibility of further contamination, probably by checking drains and manholes for signs of the spill material. And it helps to know what the spilled material is.

KNOWLEDGE ENGINEER: How can you tell what it is? [The knowledge engineer focuses the interview on the subproblem of identifying the material.]

DOMAIN EXPERT: Sometimes you can tell what the substance is by its smell. Sometimes you can tell by its color, but that's not always reliable since dyes are used a lot nowadays. Oil, however, floats on the surface and forms a silvery film, while acids dissolve completely in the water. [The expert goes on to the next subproblem.] Once you discover the type of material spilled, you can eliminate any buildings that either don't store the material at all or don't store enough of it to account for the spill.

At this point, the knowledge engineer starts using his tools to prototype the expert system. After writing some rules, a subsequent session might begin as follows:

KNOWLEDGE ENGINEER: Here are some rules that I think capture your explanation about determining the type of material spilled and elimination of possible spill sources. [Shows what he has written.] What do you think?

DOMAIN EXPERT: [After examining the rules.] Yes, that begins to capture it. Of course if the material is silver nitrate it will dissolve only partially in the water.

KNOWLEDGE ENGINEER: I see. Well, let's add that information to the knowledge base and see what it looks like. [Writes a new rule.]

The knowledge engineers repeat the cycle of collecting information from the domain experts, integrating the new information into the expert system, and bringing it back to the domain expert for evaluation and modification many times before the expert system if finished. One of the problems with working with the domain experts is that they often don't know how they reached a certain conclusion. The decisions made by human experts are often the product of extensive experience and educated guesses and are rarely the product of some conscious process. Even when a reason is given by the domain experts for their thinking, it is often not how they actually arrived at the conclusion.

In trying to decide how to teach his students how to add fractions, a friend recently interviewed a series of acquaintances. Each was a tradesman who had to add fractions as part of his work. My friend began by asking each how he would add one half and one fourth. Most of the tradesmen hemmed and hawed and, although they could quickly give the correct answer, could provide no explanation as to how they actually arrived at that answer. A few immediately replied that they changed the fractions to a common denominator and then added the numerators (the classic method taught for adding fractions). Each, however, then admitted that that was not how they had actually arrived at the answer. Only one of those could give an

adequate explanation of how they added two fractions. This is typical of what knowledge engineers must go through to collect information for an expert system.

There are two key parts to an expert system: the knowledge base and the inference engine. The knowledge base is made up of the rules the knowledge engineer has generated from the domain experts. Expert systems often end up containing more than a hundred rules and some have up to ten thousand or more. The inference engine reaches conclusions using the knowledge base and information entered by the user. The inference engine continues to ask for more information, going through the rules until it reaches a conclusion. More sophisticated expert systems also have modules for explaining their advice.

How the Program Works

Our program, VEXPERT (Vacation EXPERT), is an expert system to help you select a vacation location. It contains routines to handle two of the three functions of most expert systems. It will help you solve the problem of selecting a vacation spot, and it will learn new vacation spots you wish to add to the system. Here is a sample conversation with VEXPERT:

```
THIS PROGRAM WILL HELP YOU SELECT A
  LOCATION FOR YOUR VACATION

SELECT AN OPTION
1) RUN VEXPERT
2) ADD TO KNOWLEDGE
3) QUIT

1
JUST ANSWER EACH YES/NO QUESTION.

DO YOU WANT TO SIT IN THE SUN ON THE
  BEACH?
yes

DO YOU WISH TO VISIT A PACIFIC TROPICAL
  PARADISE?
yes

YOU SHOULD VACATION IN HAWAII



THIS PROGRAM WILL HELP YOU SELECT A
  LOCATION FOR YOUR VACATION.

SELECT AN OPTION
1) RUN VEXPERT
2) ADD TO KNOWLEDGE
3) QUIT

1
JUST ANSWER EACH YES/NO QUESTION.

DO YOU WANT TO SIT IN THE SUN ON THE
  BEACH?
yes

DO YOU WISH TO VISIT A PACIFIC TROPICAL
  PARADISE?
no

YOU SHOULD VACATION IN FLORIDA

THIS PROGRAM WILL HELP YOU SELECT A
  LOCATION FOR YOUR VACATION.

SELECT AN OPTION
1) RUN VEXPERT
2) ADD TO KNOWLEDGE
3) QUIT

1
JUST ANSWER EACH YES/NO QUESTION.

DO YOU WANT TO SIT IN THE SUN ON THE
  BEACH?
no

DO YOU WANT TO VISIT NEW ENGLAND?
no

DO YOU WANT TO GO SKIING?
yes

YOU SHOULD VACATION IN COLORADO
```

The data structure is the secret to VEXPERT's working. The knowledge base is made up of a collection of variables all having the same form. The contents of

each variable is made up of three parts. For example, in the variable below, called :MAINE.BRANCH, the first part is a list containing a question. The last two parts are words indicating either vacation suggestions or other variables with a further question. A variable name ends in ".BRANCH". The middle portion of the variable is used if the user answers "yes" to visiting New England. The answer then would be that the user should visit Maine. If the user answered "no," however, the variable :COLORADO.BRANCH would be called.

```
[[DO YOU WANT TO VISIT NEW ENGLAND?]
  MAINE  COLORADO.BRANCH]
```

Three procedures assist with extracting the three parts. QUESTION takes a variable in the knowledge base and outputs the question. YES.BRANCH will output the middle portion of the variable and NO.BRANCH outputs the last part.

```
TO QUESTION :BRANCH
OUTPUT FIRST THING :BRANCH
END

TO YES.BRANCH :BRANCH
OUTPUT FIRST BUTFIRST THING :BRANCH
END

TO NO.BRANCH :BRANCH
OUTPUT LAST THING :BRANCH
END
```

The procedure INIT initializes these variables in the knowledge base. INIT is designed to make creating other knowledge bases easier: the first variable in the built-in knowledge base, for example, is generically called KNOWLEDGE instead of something related to vacation spots. Certain statements specific to vacationing used in the program are also stored as variables. This means you don't have to hunt for the print statements in the program if you are creating a new knowledge base. Once INIT is finished, VEXPERT calls MENU, which lets the user either run the actual expert system, learn new knowledge, or quit the program.

```
TO VEXPERT
INIT
MENU
END

TO INIT
CG
CC
HT
CT
MAKE "STATEMENT [YOU SHOULD VACATION IN]
MAKE "LEARN.STATEMENT [WHAT NEW VACATION
  SPOT DO YOU WANT TO ADD?]
MAKE "LEARNING? "FALSE
MAKE "KNOWLEDGE [[DO YOU WANT TO SIT IN
  THE SUN ON THE BEACH?]
  HAWAII.BRANCH MAINE.BRANCH]
MAKE "MAINE.BRANCH [[DO YOU WANT TO VISIT
  NEW ENGLAND?]
  MAINE COLORADO.BRANCH]
MAKE "COLORADO.BRANCH [[DO YOU WANT TO GO
  SKIING?] COLORADO CHICAGO.BRANCH]
MAKE "CHICAGO.BRANCH [[DO YOU WANT TO
  VISIT A BIG CITY?] CHICAGO WISCONSIN]
MAKE "HAWAII.BRANCH [[DO YOU WISH TO
  VISIT A PACIFIC TROPICAL PARADISE?]
  HAWAII FLORIDA]
MAKE "PREVIOUS.BRANCH "KNOWLEDGE
END

TO MENU
PRINT []
PRINT [THIS PROGRAM WILL HELP YOU SELECT]
PRINT [A LOCATION FOR YOUR VACATION.]
PRINT []
PRINT [SELECT AN OPTION]
PRINT [1) RUN EXPERT]
PRINT [2) ADD TO KNOWLEDGE]
PRINT [3) QUIT]
PRINT []
MAKE "ANS FIRST READLIST
PRINT []
IF :ANS = 1 [EXPERT]
IF :ANS = 2 [LEARN]
IF :ANS = 3 [KWIT]
MENU
END
```

EXPERT starts the user on the way to selecting a place to vacation. CHOOSE.BRANCH looks at each branch to see if it is another variable or a vacation recommendation. If it is a recommendation, RESPOND is called to make the suggestion to the user. If the branch turns out to be another variable, CHOOSE.BRANCH will ask the user the yes-no question in the variable. CHOOSE.BRANCH is then recursively called with either the yes-part of the variable or the no-part, depending on how the user answers the question. Questions are asked until CHOOSE.BRANCH comes to a vacation recommendation.

```
TO EXPERT
PRINT [JUST ANSWER EACH YES/NO QUESTION.]
PRINT []
CHOOSE.BRANCH "KNOWLEDGE
END

TO CHOOSE.BRANCH :BRANCH
IF NOT NAME? :BRANCH [RESPOND :BRANCH
  STOP]
MAKE "RESPONSE ASK.YES.OR.NO QUESTION
  :BRANCH
MAKE "PREVIOUS.BRANCH :BRANCH
IF "RESPONSE = [YES] [CHOOSE.BRANCH
  YES.BRANCH :BRANCH STOP]
CHOOSE.BRANCH NO.BRANCH :BRANCH
END

TO RESPOND :BRANCH
IF :LEARNING? [EXPAND.KNOWLEDGE :BRANCH
  :PREVIOUS.BRANCH :RESPONSE
  STOP]
PRINT []
(PRINT :STATEMENT UNPACK.NAME :BRANCH)
PRINT []
END

TO ASK.YES.OR.NO :QUESTION
PRINT []
PRINT :QUESTION
MAKE "INPUT READLIST
IF :INPUT = [YES] [OUTPUT [YES]]
IF :INPUT = [NO] [OUTPUT [NO]]
PRINT []
PRINT [PLEASE TYPE "YES" OR "NO"]
OUTPUT ASK.YES.OR.NO :QUESTION
END
```

The knowledge base can be expanded using the LEARN routine from the menu. First, LEARN asks the user for the new vacation spot. Let's say you want to add VERMONT to the system. Next, LEARN calls EXPERT so that, from the user's answers to the questions in the knowledge base, LEARN can discover where to put the new vacation suggestion within the knowledge base. Recall :MAINE.BRANCH.

```
[[DO YOU WANT TO VISIT NEW ENGLAND?]
  MAINE COLORADO.BRANCH]
```

When the computer asks "DO YOU WANT TO VISIT NEW ENGLAND?" you answer "yes." Because MAINE is a suggested location and not another variable with a question, VEXPERT knows it has found where to put VERMONT. The procedure EXPAND.KNOWLEDGE is now called by EXPERT. It asks you for a question that is yes for VERMONT and no for MAINE. Perhaps you type "DO YOU WANT TO AVOID THE COAST?" A new variable is then created, called :VERMONT.BRANCH, which looks like this:

```
[[DO YOU WANT TO AVOID THE COAST?]
  VERMONT MAINE]
```

MODIFY.BRANCH is called next. This procedure is responsible for substituting VERMONT.BRANCH for MAINE in :MAINE.BRANCH, finishing off the updating of the knowledge base.

```
TO LEARN
PRINT :LEARN.STATEMENT
MAKE "NEW.ANSWER PACK.NAME READLIST
MAKE "LEARNING? "TRUE
PRINT []
PRINT [ANSWER THE FOLLOWING QUESTIONS.]
EXPERT
PRINT []
(PRINT UNPACK.NAME :NEW.ANSWER [NOW ADDED
  TO KNOWLEDGE])
MAKE "LEARNING? "FALSE
END
```

```
TO EXPAND.KNOWLEDGE :BRANCH
  :PREVIOUS.BRANCH :YES.NO
PRINT []
PRINT [PLEASE TYPE IN A YES-NO QUESTION]
PRINT [ WHOSE ANSWER IS YES FOR]
(PRINT UNPACK.NAME :NEW.ANSWER [AND NO
  FOR]
  UNPACK.NAME :BRANCH )
PRINT []
MAKE "QUESTION READLIST
MAKE WORD :NEW.ANSWER ".BRANCH FPUT
  :QUESTION
  SENTENCE :NEW.ANSWER :BRANCH
MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
  :YES.NO
END

TO MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
  :YES.NO
MAKE "YES.PART YES.BRANCH
  :PREVIOUS.BRANCH
MAKE "NO.PART NO.BRANCH :PREVIOUS.BRANCH
IFELSE :YES.NO = [YES] [MAKE "YES.PART
  WORD :NEW.ANSWER ".BRANCH ]
  [MAKE "NO.PART WORD :NEW.ANSWER
  ".BRANCH ]
MAKE "QUESTION.PART QUESTION
  :PREVIOUS.BRANCH
MAKE :PREVIOUS.BRANCH FPUT :QUESTION.PART
  SENTENCE :YES.PART :NO.PART
END
```

Two other procedures assist VEXPERT. They combine the names of vacation spots that might be more than one word into a single word. Recall that the variables in the knowledge base contain only single words for the yes- and no-parts. If someone wished to enter the Virgin Islands as a vacation suggestion, PACK.NAME will change it to VIRGIN.ISLANDS by substituting a period for the space. This transforms "VIRGIN ISLANDS" into a single word for storage in a variable. When VEXPERT needs to print out this information, UNPACK.NAME takes out the period and puts a space back in so it reads normally.

```
TO PACK.NAME :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF (COUNT :LIST) = 1 [OUTPUT FIRST :LIST]
OUTPUT (WORD FIRST :LIST ". PACK.NAME
  BUTFIRST :LIST)
END

TO UNPACK.NAME :WORD
IF NOT MEMBER? ". :WORD [OUTPUT :WORD]
IF (FIRST :WORD) = ". [OUTPUT WORD CHAR
  32 UNPACK.NAME BUTFIRST :WORD]
OUTPUT WORD (FIRST :WORD) (UNPACK.NAME
  BUTFIRST :WORD)
END
```

Possible Extensions and Expansions

- VEXPERT has no way to save new knowledge entered. If you took the time to put in ten new vacation spots, they would be lost when you ended the program or turned off the machine. Modify the program so it will save new data. It might save automatically when new data is entered, or it might ask users if they wants to save new data before quitting the program.
- Create a new knowledge base. Don't forget to change the statements in the program.
- Create a file management system so that VEXPERT can call up any of the various knowledge bases. You could save the knowledge bases separate from the program shell.
- Modify VEXPERT so it can handle questions with answers other than yes or no. Perhaps it could ask what temperature range you would like to visit and then could process a range of answers. There are lots of applications for this ability; medical expert systems make decisions based on people's age, and an animal expert might ask how many legs the animal has.

- Build in a explanation function ("why" and "how"). If you asked VEXPERT why it asked you a particular question, it might say that it is trying to decide between two spots and then give you the name of those spots. If VEXPERT suggests a vacation spot and you ask how it came to that suggestion, it could reply with your answers to the questions you were asked.

VEXPERT
The Program
LogoWriter Version

```
TO VEXPERT
INIT
MENU
END

TO INIT
CG
CC
HT
CT
MAKE "STATEMENT [YOU SHOULD VACATION IN]
MAKE "LEARN.STATEMENT [WHAT NEW VACATION
  SPOT DO YOU WANT TO ADD?]
MAKE "LEARNING? "FALSE
MAKE "KNOWLEDGE [[DO YOU WANT TO SIT IN
  THE SUN ON THE BEACH?]
  HAWAII.BRANCH MAINE.BRANCH]
MAKE "MAINE.BRANCH [[DO YOU WANT TO VISIT
  NEW ENGLAND?]
  MAINE COLORADO.BRANCH]
MAKE "COLORADO.BRANCH [[DO YOU WANT TO GO
  SKIING?]COLORADO CHICAGO.BRANCH]
MAKE "CHICAGO.BRANCH [[DO YOU WANT TO
  VISIT A BIG CITY?]
  CHICAGO WISCONSIN]
MAKE "HAWAII.BRANCH [[DO YOU WISH TO
  VISIT A PACIFIC TROPICAL PARADISE?]
  HAWAII FLORIDA]
MAKE "PREVIOUS.BRANCH "KNOWLEDGE
END

TO MENU
PRINT []
PRINT [THIS PROGRAM WILL HELP YOU SELECT]
PRINT [A LOCATION FOR YOUR VACATION.]
PRINT []
PRINT [SELECT AN OPTION]
PRINT [1) RUN EXPERT]
PRINT [2) ADD TO KNOWLEDGE]
PRINT [3) QUIT]
PRINT []
MAKE "ANS FIRST READLIST
PRINT []
IF :ANS = 1 [EXPERT]
IF :ANS = 2 [LEARN]
IF :ANS = 3 [KWIT]
MENU
END

TO EXPERT
PRINT [JUST ANSWER EACH YES/NO QUESTION.]
PRINT []
CHOOSE.BRANCH "KNOWLEDGE
END

TO CHOOSE.BRANCH :BRANCH
IF NOT NAME? :BRANCH [RESPOND :BRANCH
  STOP]
MAKE :RESPONSE ASK.YES.OR.NO QUESTION
  :BRANCH
MAKE "PREVIOUS.BRANCH :BRANCH
IF "RESPONSE = [YES] [CHOOSE.BRANCH
  YES.BRANCH :BRANCH STOP]
CHOOSE.BRANCH NO.BRANCH :BRANCH
END

TO RESPOND :BRANCH
IF :LEARNING? [EXPAND.KNOWLEDGE :BRANCH
  :PREVIOUS.BRANCH :RESPONSE STOP]
PRINT []
(PRINT :STATEMENT UNPACK.NAME :BRANCH)
PRINT []
END

TO ASK.YES.OR.NO :QUESTION
PRINT []
PRINT :QUESTION
MAKE "INPUT READLIST
IF :INPUT = [YES] [OUTPUT [YES]]
IF :INPUT = [NO] [OUTPUT [NO]]
PRINT []
PRINT [PLEASE TYPE "YES" OR "NO"]
OUTPUT ASK.YES.OR.NO :QUESTION
END
```

```
TO QUESTION :BRANCH
OUTPUT FIRST THING :BRANCH
END

TO YES.BRANCH :BRANCH
OUTPUT FIRST BUTFIRST THING :BRANCH
END

TO NO.BRANCH :BRANCH
OUTPUT LAST THING :BRANCH
END

TO LEARN
PRINT :LEARN.STATEMENT
MAKE "NEW.ANSWER PACK.NAME READLIST
MAKE "LEARNING? "TRUE
PRINT []
PRINT [ANSWER THE FOLLOWING QUESTIONS.]
EXPERT
PRINT []
(PRINT UNPACK.NAME :NEW.ANSWER [NOW ADDED
  TO KNOWLEDGE])
MAKE "LEARNING? "FALSE
END

TO EXPAND.KNOWLEDGE :BRANCH
  :PREVIOUS.BRANCH :YES.NO
PRINT []
PRINT [PLEASE TYPE IN A YES-NO QUESTION]
PRINT [ WHOSE ANSWER IS YES FOR]
(PRINT UNPACK.NAME :NEW.ANSWER [AND NO
  FOR] UNPACK.NAME :BRANCH )
PRINT []
MAKE "QUESTION READLIST
MAKE WORD :NEW.ANSWER ".BRANCH FPUT
  :QUESTION SENTENCE :NEW.ANSWER :BRANCH
MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
  :YES.NO
END

TO MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
  :YES.NO
MAKE "YES.PART YES.BRANCH
  :PREVIOUS.BRANCH
MAKE "NO.PART NO.BRANCH :PREVIOUS.BRANCH
IFELSE :YES.NO = [YES] [MAKE "YES.PART
  WORD :NEW.ANSWER ".BRANCH ]
  [MAKE "NO.PART WORD :NEW.ANSWER
  ".BRANCH ]
MAKE "QUESTION.PART QUESTION
  :PREVIOUS.BRANCH
MAKE :PREVIOUS.BRANCH FPUT :QUESTION.PART
  SENTENCE :YES.PART :NO.PART
END

TO PACK.NAME :LIST
IF EMPTY? :LIST [OUTPUT :LIST]
IF (COUNT :LIST) = 1 [OUTPUT FIRST :LIST]
OUTPUT (WORD FIRST :LIST ". PACK.NAME
  BUTFIRST :LIST)
END

TO UNPACK.NAME :WORD
IF NOT MEMBER? ". :WORD [OUTPUT :WORD]
IF (FIRST :WORD) = ". [OUTPUT WORD CHAR
  32 UNPACK.NAME BUTFIRST :WORD]
OUTPUT WORD (FIRST :WORD) (UNPACK.NAME
  BUTFIRST :WORD)
END

TO KWIT
PRINT []
PRINT [THANKS FOR USING YOUR]
PRINT [VACATION EXPERT]
PRINT []
STOPALL
END
```

VEXPERT
The Program
Terrapin Logo PLUS Version

```
TO VEXPERT
 INIT
 MENU
END

TO INIT
  CLEARTEXT
  TEXTSCREEN
  MAKE "STATEMENT [YOU SHOULD VACATION IN]
  MAKE "LEARN.STATEMENT [WHAT NEW VACATION
   SPOT DO YOU WANT TO ADD?]
  MAKE "LEARNING? "FALSE
  MAKE "KNOWLEDGE [[DO YOU WANT TO SIT IN
   THE SUN ON THE BEACH?]
   HAWAII.BRANCH MAINE.BRANCH]
  MAKE "MAINE.BRANCH [[DO YOU WANT TO
   VISIT NEW ENGLAND?]
   MAINE COLORADO.BRANCH]
  MAKE "COLORADO.BRANCH [[DO YOU WANT TO
   GO SKIING?]
   COLORADO CHICAGO.BRANCH]
  MAKE "CHICAGO.BRANCH [[DO YOU WANT TO
   VISIT A BIG CITY?]
   CHICAGO WISCONSIN]
  MAKE "HAWAII.BRANCH [[DO YOU WISH TO
   VISIT A PACIFIC TROPICAL PARADISE?]
   HAWAII FLORIDA]
  MAKE "PREVIOUS.BRANCH "KNOWLEDGE
END

TO MENU
 PRINT []
 PRINT [THIS PROGRAM WILL HELP YOU
  SELECT]
 PRINT [A LOCATION FOR YOUR VACATION.]
 PRINT []
 PRINT [SELECT AN OPTION]
 PRINT [1) RUN EXPERT]
 PRINT [2) ADD TO KNOWLEDGE]
 PRINT [3) QUIT]
 PRINT []
 MAKE "ANS FIRST REQUEST
 PRINT []
 IF :ANS = 1 THEN EXPERT
 IF :ANS = 2 THEN LEARN
 IF :ANS = 3 THEN KWIT
 MENU
END

TO EXPERT
 PRINT [JUST ANSWER EACH YES/NO
  QUESTION.]
 PRINT []
 CHOOSE.BRANCH "KNOWLEDGE
END

TO CHOOSE.BRANCH :BRANCH
 IF NOT THING? :BRANCH THEN RESPOND
  :BRANCH STOP
 MAKE "RESPONSE ASK.YES.OR.NO QUESTION
  :BRANCH
 MAKE "PREVIOUS.BRANCH :BRANCH
 IF :RESPONSE = [YES] THEN CHOOSE.BRANCH
  YES.BRANCH :BRANCH STOP
 CHOOSE.BRANCH NO.BRANCH :BRANCH
END

TO RESPOND :BRANCH
 IF :LEARNING? THEN EXPAND.KNOWLEDGE
  :BRANCH :PREVIOUS.BRANCH :RESPONSE
  STOP
 PRINT []
 (PRINT :STATEMENT UNPACK.NAME :BRANCH)
 PRINT []
END

TO ASK.YES.OR.NO :QUESTION
 PRINT []
 PRINT :QUESTION
 MAKE "INPUT REQUEST
 IF :INPUT = [YES] THEN OUTPUT [YES]
 IF :INPUT = [NO] THEN OUTPUT [NO]
 PRINT []
 PRINT [PLEASE TYPE "YES" OR "NO"]
 OUTPUT ASK.YES.OR.NO :QUESTION
END

TO QUESTION :BRANCH
 OUTPUT FIRST THING :BRANCH
END

TO YES.BRANCH :BRANCH
 OUTPUT FIRST BUTFIRST THING :BRANCH
END

TO NO.BRANCH :BRANCH
 OUTPUT LAST THING :BRANCH
END
```

```
TO LEARN
  PRINT :LEARN.STATEMENT
  MAKE "NEW.ANSWER PACK.NAME REQUEST
  MAKE "LEARNING? "TRUE
  PRINT []
  PRINT [ANSWER THE FOLLOWING QUESTIONS.]
  EXPERT
  PRINT []
  (PRINT UNPACK.NAME :NEW.ANSWER [NOW
   ADDED TO KNOWLEDGE])
  MAKE "LEARNING? "FALSE
END

TO EXPAND.KNOWLEDGE :BRANCH
  :PREVIOUS.BRANCH :YES.NO
  PRINT []
  PRINT [PLEASE TYPE IN A YES-NO QUESTION]
  PRINT [ WHOSE ANSWER IS YES FOR]
  (PRINT UNPACK.NAME :NEW.ANSWER [AND NO
   FOR] UNPACK.NAME :BRANCH )
  PRINT []
  MAKE "QUESTION REQUEST
  MAKE WORD :NEW.ANSWER ".BRANCH FPUT
   :QUESTION SENTENCE :NEW.ANSWER :BRANCH
  MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
   :YES.NO
END

TO MODIFY.BRANCH :BRANCH :PREVIOUS.BRANCH
  :YES.NO
  MAKE "YES.PART YES.BRANCH
  :PREVIOUS.BRANCH
  MAKE "NO.PART NO.BRANCH
  :PREVIOUS.BRANCH
  IF :YES.NO = [YES] THEN MAKE "YES.PART
  WORD :NEW.ANSWER ".BRANCH
  ELSE MAKE "NO.PART WORD :NEW.ANSWER
  ".BRANCH
  MAKE "QUESTION.PART QUESTION
  :PREVIOUS.BRANCH
  MAKE :PREVIOUS.BRANCH FPUT
  :QUESTION.PART SENTENCE :YES.PART
  :NO.PART
END

TO PACK.NAME :LIST
  IF EMPTY? :LIST THEN OUTPUT :LIST
  IF (COUNT :LIST) = 1 THEN OUTPUT FIRST
   :LIST
  OUTPUT (WORD FIRST :LIST ". PACK.NAME
   BUTFIRST :LIST)
END

TO UNPACK.NAME :WORD
  IF NOT MEMBER? ". :WORD THEN OUTPUT
   :WORD
  IF (FIRST :WORD) = ". THEN OUTPUT WORD
   CHAR 32 UNPACK.NAME BUTFIRST :WORD
  OUTPUT WORD (FIRST :WORD) (UNPACK.NAME
   BUTFIRST :WORD)
END

TO KWIT
  PRINT []
  PRINT [THANKS FOR USING YOUR]
  PRINT [VACATION EXPERT]
  PRINT []
  TOPLEVEL
END
```

Annotated Bibliography

This reference list includes additional resources that you may find interesting.

Abelson, H. (1982). *Logo for the Apple II.* Peterborough, NH: BYTE/McGraw-Hill.

A terrific introduction to Logo for people who are already experienced computer users. Includes a discussion of the learning program ANIMALS and the conversationalist DOCTOR.

Albus, J. S. (1981). *Brains, behavior and robotics.* Peterborough, NH: BYTE/McGraw-Hill.

The first half of the book discusses how the systems of the brain work; the second half examines how scientists are creating machines to emulate these systems.

Brand, S. (1987). *The media lab: Inventing the future at MIT.* New York: Penguin Books.

This book provides a fascinating look into the people, facilities, and projects of MIT's Media Lab. Each high-tech project is like something from a science fiction story but exists in Cambridge, Massachusetts. The emerging technologies will fascinate everyone.

Chabris, C. F. (1987). *Artificial intelligence and Turbo Pascal.* Homewood, IL: Dow Jones-Irwin.

Includes a Turbo Pascal data disk of programs related to the topics covered in the book. The book thoroughly examines several fields of artificial intelligence but does not describe how the Pascal programs work. Knowledge of Turbo Pascal is not necessary for the reader to get a lot from the text. An excellent reference for the initiated student of AI.

The Computer Museum. (1987, Summer/Fall). *The Computer Museum report.* Boston, MA: The Computer Museum.

An introduction to the fantastic AI exhibit at The Computer Museum. The Report outlines the creation of the exhibit, provides an overview of artificial intelligence, and supplies the reader with a terrific timeline of artificial intelligence and robotics. Although the report is full of interesting pictures and sidebars, it is no substitute for visiting the exhibit.

Friedman, D. P. (1987). *The little LISPer.* Cambridge, MA: MIT Press.

A classic introduction to LISP, one of the two most widely used languages for studying AI. Logo is a dialect of LISP. An engaging book that explains difficult programming concepts in a fast-paced and humorous dialog.

Friendly, M. (1988). *Advanced Logo, a language for learning.* Hillsdale, NJ: Lawrence Erlbaum Associates.

My favorite Logo book. A thorough look at the many features of Logo. The book is written in an informal,

conversational tone and includes hundreds of procedures. The book includes listings for programs related to knowledge representation, robot simulation (BLOCKS), pattern recognition, and language processing.

Goldenberg, E. P., & Feurzeig, W. (1987). *Exploring language with Logo*. Cambridge, MA: MIT Press.

The first book published in MIT's *Exploring with Logo* series, this book "invites the reader to participate actively in the play and work of the linguist." It is one of the first books to take full advantage of Logo's language capabilities. The reader discovers the elegance and complexities of language through experiments and exercises in Logo. Don't read this book without having your computer and Logo close at hand.

Harper, D. (1989). *Logo: Theory & practice*. Pacific Grove, CA: Brooks/Cole Publishing Company.

This book is largely a collection of articles from the *Logo Exchange* and is probably the best (in my opinion) introductory book on Logo for teachers. It is a wonderful collage of philosophy, teaching ideas, puzzles, and more.

Hartnell, T. (1985). *Exploring artificial intellegence on your Apple II*. New York: Bantam Books.

A wonderful book that explores AI through the programming language BASIC. Good discussions of AI topics and excellent programs illustrating fields of AI. Well written for the computer hobbyist.

Harvey, B. (1987). *Computer science Logo style*. (Vol. I, II, & III). Cambridge, MA: MIT Press.

These books represent a new approach to teaching computer science in high schools and colleges by using Logo. Volume I focuses on the details of program design and Logo's syntax and capabilities. Volume II is a project book describing some more advanced programming techiques and interesting Logo projects. Volume III introduces some of the advanced programming topics that make up a college-level computer science curriculum.

Krutch, J. (1986). *Experiments in artificial intellegence for microcomputers*. Indianapolis, IN: Howard W. Sams & Co.

Like Hartnell's book, a collection of discussions of AI topics with BASIC programs as illustrations.

Mishkoff, H. C. (1988). *Understanding artificial intelligence*. Indianapolis, IN: Howard W. Sams & Co.

This book would make a good AI text for a nonprogramming course. It examines the major fields of AI and includes discussions of many of the commercially available AI programs and companies. The book is full of good examples, illustrations, and sidebars.

Peat, F. D. (1988). *Artificial intelligence: How machines think*. New York: Baen Books.

My favorite AI book. It thoroughly covers the many fields of AI and is written in a flowing, easy-to-read style. The reader need not be either a programmer or a computer person to understand and benefit from this excellent book. The book includes sample runs of many of the historic AI programs.

Rosenthal, L. (1983, January). Friendly interface, building computers with hearts. *Science Digest.*

An excellent discussion of some of the natural language and expert system programs that researchers are creating. The article does not cover the programming involved but gives lots of examples of the programs working. Very interesting reading.

Simons, G. (1984). *Computer bits and pieces.* New York: Penguin Books.

A terrific collection of interesting computer programs and curiosities. Although they are organized by area of application, the book can be opened to any page to find some interesting use of the computer.

Terrapin, Inc. (1988). *The Terrapin Logo PLUS reference manual.* Portland, ME: Terrapin, Inc.

This manual contains a variety of sample programs as well as reference material about Logo PLUS.

Walter, R. (1987). *The secret guide to computers.* (Vol. I, II, & III). Somerville, MA: Russ Walter Publishers.

The author takes a humorous and refreshing stab at clarifying computers to a techniphobic audience. Russ Walter aimed the book primarily toward college students, support staff, and business people and (although he doesn't really understand educational technology) succeeds in presenting the basics in an interesting way. Anyone who gets a bit anxious when they face that machine on their desk should consider taking a look at these books.

Weston, D. (1985). *The second Logo book: Advanced techniques in Logo.* Glenview, IL: Scott, Foresman and Company.

As the title implies, a terrific book for readers interested in going beyond turtle graphics. Topics include recursion, words and lists, property lists, trees, and pattern matching. The book includes a healthy variety of tool procedures and an interesting version of BLOCKS.

Wiencek, H. (1987). *The world of LEGO toys.* New York: Harry N. Abrams, Inc.

A history of the successful toys, LEGO building bricks. The book not only describes the history of the LEGO brick but also discusses its uses by artists, architects and teachers, as well as children. There are also wonderful sections on the philosophy behind the brick and the Danish amusement park, LEGOLAND. The book is fascinating reading, full of interesting pictures.

Wurman, R. S. (1989). *Information anxiety*. New York: Doubleday.

"Information Anxiety is produced by the ever-widening gap between what we understand and what we think we should understand. It is the black hole between data and knowledge, and it happens when information doesn't tell us what we want or need to know." Mr. Wurman's innovative book addresses some of society's changes that we need to acknowledge and advises us on how we can begin to deal with the overload of information. Every teacher should read *Information Anxiety*.

Index

A

AI *(see Artificial Intelligence)*
algorithms 44
alpha-beta algorithm 20,23
Artificial Intelligence 1, 19, 53, 54
  understanding 53, 54
Asimov, Isaac 10
assembly language 33

B

backgammon 19
Berliner, Hans 19
binary 33
BUGGY 109

C

Caduceus 109
Capek, Karel 9-10
Carnegie-Mellon University 11
CART 11
checkers 19, 20
chess 9, 19, 20
Chest Pain 108
COGEN 106
Colby, Kenneth 45
computer algebra systems 109
computer memory size 34-35, 95
computer processor speed 34-35, 95
conversation 95-96
conversationalist 43-46
CRYSALIS 106
Cult 53
CYRUS 69

D

Dartmouth Conference 19
DENDRAL 106
DERIVE 109
Digital Equipment Corporation 110
Digitalis Therapy Advisor 108-109
domain experts 110-112
Draughtsman 9

E

Eliza 44-45, 46
EMYCIN 108
evaluation procedure 20
expert, human 19
expert system 19, 105-112
  chemistry 105-106
  education 109
  mathematics 109-110
  medicine 106-109
expert system shell 108

F

Fischer Technik 12

G

Golem 9
grammar 33, 96-97
Guidon 109

H

Hero of Alexandria 9
heuristics 44
high level language 33-34, 35
Hypothetical Syllogism 69-70

I

ICAI 109
inference engine 69,112

J

Jacquet-Droz, Henri-Louis 9

Jacquet-Droz, Pierre 9

K

Kemeny, John 45
keyword analysis 96
knowledge base 112-113
knowledge engineers 110-112
knowledge representation 81-83

L

language 33-35
LEGO 12
Levy, David 19, 22
logic 69
logic, valid 69, 70
Logo 2
Logo primitive
  dot? 13
  run 13-14
LogoWriter 1, 2, 4, 5, 6, 12
Luigi 43

M

machine language 33, 34-35
machine translation systems 53
Macsyma 109
Maple 109
MathematicaL 109
McCarthy, John 19
microworld 96, 97
minimaxing procedure 20, 21
Minsky, Marvin 19
Musician 9
MYCIN 106-108, 109

N

natural language 33, 34, 44, 53, 95-98

P

Parry 45-46
parsing 96
Poe, Edgar Allen 9
properties 83
  inheritance 83
PROSPECTOR 110
pruning 21-22
PUFF 108

R

Rabbi Loew 9
REACT 106
robot 9, 43
  arms 10-11
  general 11
  industrial 10, 12
  legged 11-12
  toy 12-13
  Three Laws of Robots 10
  turtle 12-13
  vision 11
Rogerian psychology 44
Rossum's Universal Robots 9-10

S

Samuel, Arthur 19, 20
Scribe 9
searching 21
searching, ply 22
sensor 12-13
  touch 11
  vision 11
SHAKEY 11
Shannon, Claude 19
SHRDLU 54-55
Stanford Research Institute 11
Steamer 109
symbolic manipulators 109
SYNCHEM 106
Systran 53

T

Terrapin Logo 2, 12
Thaddeus 9
Token/Type Distinction Problem 83

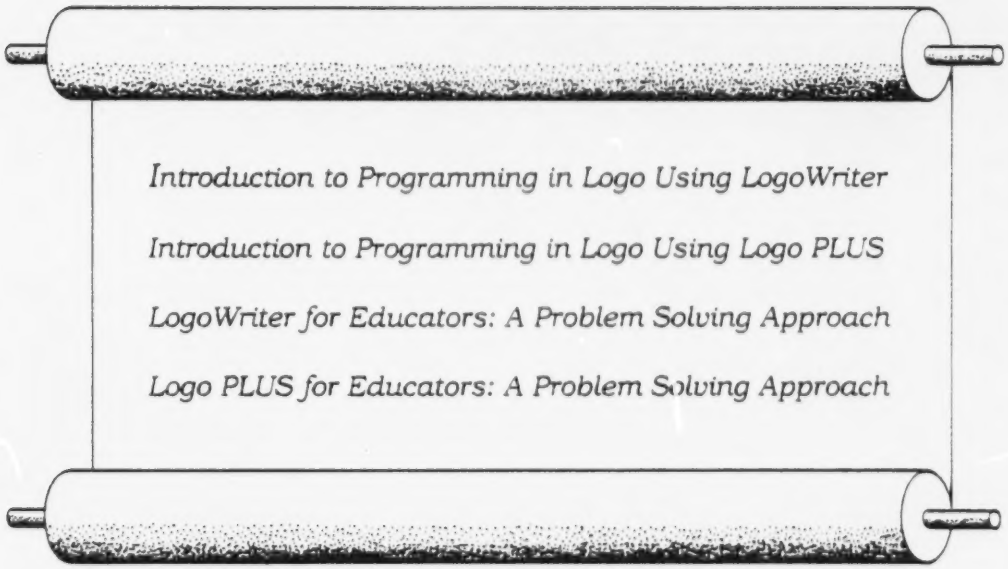
tree 21
Turing Test 43-44, 46
Turning, Alan 43

W
Weizenbaum, Joseph 44-45
Winograd, Terry 54

X
XCON 110

130

Look at our Logo list!



Introduction to Programming in Logo Using LogoWriter

Introduction to Programming in Logo Using Logo PLUS

LogoWriter for Educators: A Problem Solving Approach

Logo PLUS for Educators: A Problem Solving Approach

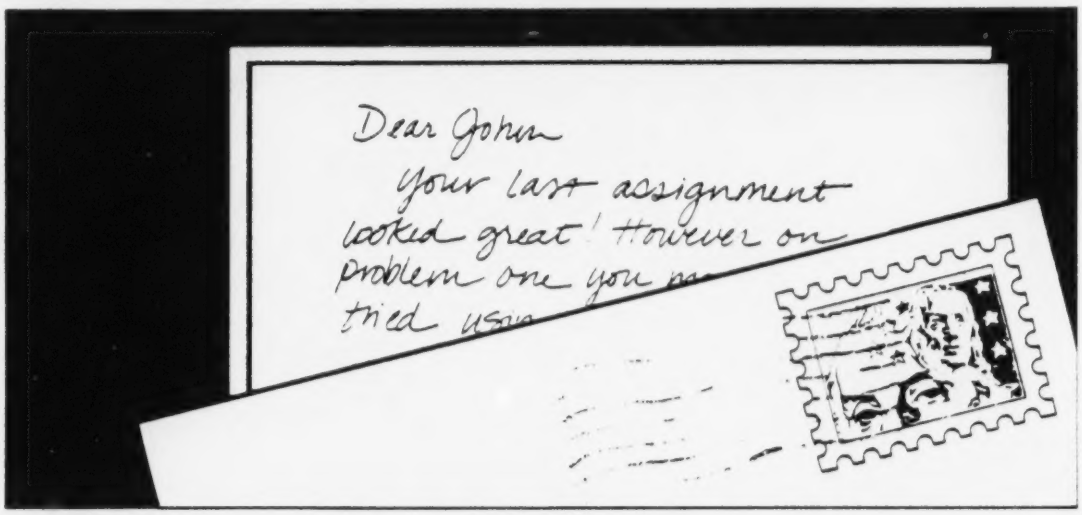
Logo users at all levels benefit from these ISTE selections.

The *Introduction to Programming* books, written by Sharon Yoder, provide beginners with a Logo base to build on and experienced users with a reference to rely on. Both are excellent resources for teacher training or introductory computer science classes.

New from ISTE, *LogoWriter (Logo PLUS) for Educators: A Problem Solving Approach* takes Logo learning to new depths. The focus is entirely on learning and practicing general problem solving skills while using Logo. Great for beginning programming experience. Appendices include keystroke summaries, turtle shape pictures, and a quick reference card. Written by Dave Moursund and Sharon Yoder.

To order contact: ISTE 1787 Agate Street, Eugene, OR 97403-1923
503/346-4414

Finally, a long distance relationship that won't break your heart.



ISTE offers ten *Independent Study* courses that get to the heart of learning.

Each course thoroughly covers the title material and is designed to provide staff development and leadership training. You correspond directly with the course's instructor by mail, and can receive graduate credit through the Oregon State System of Higher Education.

Classes offered this year are:

- Introduction to Logo for Educators (available for LogoWriter or Logo PLUS)
- Fundamentals of Computers in Education
- Long Range Planning for Computers in Schools
- Computers in Mathematics Education
- Computers and Problem Solving
- Introduction to AppleWorks for Educators
- Computers in Composition
- Effective Inservice for Instructional Use of Computers in Education
- Computer Applications for Educators: An Introduction to Microsoft Works
- Telecommunication and Information Access in Education

Register for classes independently or with a group. Districts enrolling six or more teachers receive a fee reduction for each person enrolled.

Courses range in price for 3-4 quarter-hours of graduate credit. You have one year to complete your course.

Start a great long distance relationship today with an ISTE *Independent Study* Course.

Request an *Independent Study* course brochure. Write or call:

ISTE, Independent Study Course Dept.
1787 Agate St., Eugene, OR 97403-1923
503/346-4414

Add another brick to your Logo base.

ISTE's Special Interest Group for Logo Educators (SIGLogo) provides you with a broad Logo foundation.

Build your knowledge with the latest information on Logo research, resources, and methods. Expand your Logo alternatives through the exchange of ideas, concepts, and techniques.

Both novice and experienced Logo users will find constructive uses in their SIGLogo membership.

Join SIGLogo now. Membership includes eight issues of *Logo Exchange*, the SIGLogo journal. Members are invited to participate in local, regional, and national meetings and to contribute to the flow of ideas through the *Logo Exchange*. SIGLogo membership rates are available by contacting ISTE.

ISTE, 1787 Agate St., Eugene, OR 97403-1923
503/346-4414

Their inquiring minds want to know.

And you'll help them discover how to find the answers. By using ***Teaching Thinking Skills with Databases*** in your classroom, you'll challenge students to develop a mind of their own.

Designed for Grades 4-8, this step-by-step guide by Jim Watson gives you the opportunity to impact your students' cognitive development through the use of databases.

Teaching Thinking Skills with Databases contains 14 data files and 46 worksheet and transparency masters. Teach with databases in any subject using *AppleWorks®* or *FrEdBase*.

Use ***Teaching Thinking Skills with Databases***... because they want to know.

For more information contact: ISTE
1787 Agate St., Eugene, OR 97403-1923
503/346-4414

Telecommunications: Make the connection.

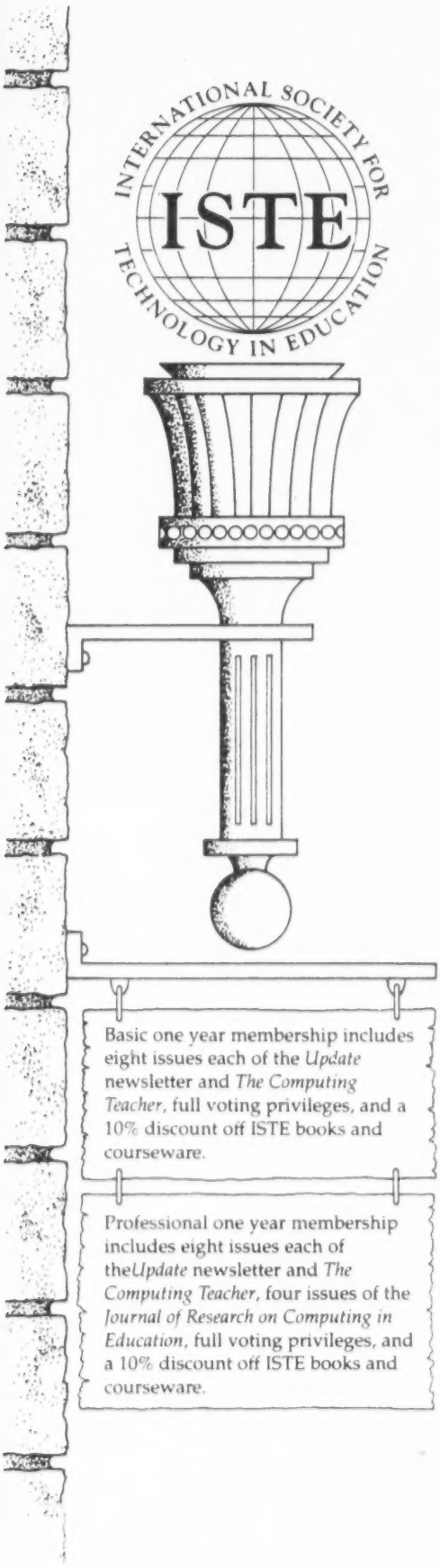
Whether you want to hook up with a teacher in Kenya, or a teacher across town, ISTE's *Telecommunications in the Classroom* will help you make the connection.

Authors Chris Clark, Barbara Kurshan, Sharon Yoder, and teachers around the world have done your homework in *Telecommunications in the Classroom*. The book details what telecommunications is, how to apply it in your classroom, what hardware and software you'll need, and what services are available. *Telecommunications in the Classroom* also includes a glossary of telecommunications terms and exemplary lesson plans from K-12 teachers.

Telecommunications in the Classroom is an affordable, informative resource for workshops, classes, and personal use.

Make your connection today with ISTE's Telecommunications in the Classroom

ISTE
1787 Agate Street, Eugene, OR 97403-1923
503/346-4414



The ***International Society for Technology in Education*** touches all corners of the world. As the largest international non-profit professional organization serving computer using educators, we are dedicated to the improvement of education through the use and integration of technology.

Drawing from the resources of committed professionals worldwide, ISTE provides information that is always up-to-date, compelling, and relevant to your educational responsibilities.

Periodicals, books and courseware, *Special Interest Groups, Independent Study* courses, professional committees, and the Private Sector Council all strive to help enhance the quality of information you receive.

Rely on ISTE support:

- *The Computing Teacher* draws on active and creative K-12 educators to provide feature articles and carefully selected columns.
- The *Update* newsletter reaches members with information on the activities of ISTE and its affiliates.
- The *Journal of Research on Computing in Education* comes out with articles on original research project descriptions and evaluations, the state of the art, and theoretical essays that define and extend the field of educational computing.
- Books and courseware enhance teaching materials for K-12 and higher education.
- Professional Committees develop and monitor policy statements on software use, ethics, preview centers, and legislative action.
- The Private Sector Council promotes cooperation between educational technology professionals, manufacturers, publishers, and other private sector organizations.

It's a big world, but with the joint efforts of educators like yourself, ISTE brings it closer. Be a part of the international sharing of educational ideas and technology. Join ISTE.

Join today, and discover how ISTE puts you in touch with the world.

Basic one year membership includes eight issues each of the *Update* newsletter and *The Computing Teacher*, full voting privileges, and a 10% discount off ISTE books and courseware.

Professional one year membership includes eight issues each of the*Update* newsletter and *The Computing Teacher*, four issues of the *Journal of Research on Computing in Education*, full voting privileges, and a 10% discount off ISTE books and courseware.

ISTE
1787 Agate Street, Eugene, OR 97403-1923
503/346-4414

BEST COPY AVAILABLE